# Emulation of IoT Devices

**Alli Mäkinen**

**School of Electrical Engineering**

Thesis submitted for examination for the degree of Master of Science in Technology.
Espoo 4.11.2016

**Thesis supervisors:**

Prof. Raimo Kantola

**Thesis advisor:**

Jaime Jimenez

**Aalto University**
**School of Electrical Engineering**

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Alli Mäkinen

Title: Emulation of IoT Devices

Date: 4.11.2016 Language: English Number of pages: 9+66

Department of Communications and Networking

Professorship:

Supervisor: Prof. Raimo Kantola

Advisor: Jaime Jimenez

A growing number of real life objects connect to the Internet forming an Internet of Things (IoT). In this concept, devices, such as sensors and actuators, control the physical environment generating a mass of data that can be used in applications and services. They are typically constrained in memory and power and thus, they need to be as lightweight as possible. Moreover, as the number of Internet-connected devices is expected to grow to billions, the technical implementations need to be scalable. This introduces a problem; managing such a large amount of devices as well as testing the different IoT scenarios may be cumbersome with existing physical testbeds, which require a lot of configuring and lack scalability.

This thesis proposes the design and implementation of emulated virtual devices using IoT spesific protocols and data models, such as CoAP, LWM2M and IPSO Objects. As device management is an important aspect of IoT, these devices are implemented to communicate with the management server through LWM2M interfaces in addition to communicating to each other. The emulated devices consist of virtual sensors and actuators represented as IPSO objects, which can be used to sense the simulated environment or control it with simple operations. Moreover, two use cases are defined and presented to create appropriate device logic. The virtualization of the devices is implemented by using Docker containers. They enable scaling the devices to hundreds, which is a key feature of the emulator.

The design of the emualor follows CoAP and LWM2M specifications, which define the set of necessary functionalities and rules for the implementation. At the end of this thesis, the emulator is evaluated by comparing it to the initial design requirements along with scalability and bandwidth usage tests. Finally, future work for improving the emulator is presented.

Keywords: Internet of Things, CoAP, LWM2M, IPSO objects, emulation, virtual devices, device management

AALTO-YLIOPISTO
SÄHKÖTEKNIIKAN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Alli Mäkinen

Työn nimi: Emulation of IoT Devices

Päivämäärä: 4.11.2016 Kieli: Englanti Sivumäärä: 9+66

Tietoliikenne- ja Tietoverkkotekniikan laitos

Professuuri:

Työn valvoja: Prof. Raimo Kantola

Työn ohjaaja: Jaime Jimenez

Internet-verkko on nopeasti laajentunut laitteisiin, jotka voivat mitata ja ohjata ympäristöään Internet-yhteyden välityksellä muodostaen tavaroiden internetin, Internet of Things (IoT). Tällaisilla laitteilla, kuten sensoreilla, on yleensä rajallisesti muistia, tehoa ja kapasiteettia tiedonkäsittelyyn. Täsä syystä onkin tärkeää, että ne ovat tekniseltä toteutukseltaan mahdollisimman kevyitä. Lisäksi IoT laitteiden määrän on ennustettu kasvavan miljardeihin, mikä tarkoittaa sitä, että teknisten toutusten on oltava myös skaalautuvia. Valtavan laitemäärän hallinta sekä erilaisten IoT-skenaarioiden testaaminen on kuitenkin hyvin vaivalloista fyysisessä testiympäristössä, erityisesti heikon skaalautuvuutensa vuosi.

Tämä diplomityö esittää ja toteuttaa ratkaisuksi emulaattorin, jolla voi emuloida useita virtuaalisia laitteita käyttäen IoT protokollia ja datamalleja, kuten CoAP- ja LWM2M-protokollia sekä IPSO-objekteja. Koska laitehallinta on olennainen osa IoT-konseptia, virtuaaliset laitteet on toteutettu niin, että ne voivat paitsi kommunikoida keskenään, niitä voi myös hallita hallintapalvelimen kautta LWM2M-opraatioita käyttäen. Laitteet koostuvat virtuaalisista sensoreista ja kytkimistä, joita mallinnetaan IPSO-objekteilla. Niiden avulla dataa voidaan kerätä ja lähettää simuloidussa ympäristössä. Lisäksi, työssä esitellään kaksi use casea, joihin toteutettu laitelogiikka pohjautuu. Virtualisointi tapahtuu Docker-platformin avulla, joka mahdollistaa laitteiden skaalaamisen satoihin.

Emulaattorin toteutus pohjautuu CoAP- ja LWM2M-standardeihin, jotka määrittävät sallitut toiminnallisuudet ja operaatiot. Diplomityön lopussa emulaattori arvioidaan toteutuneiden suunnitteluvaatimusten sekä tehtyjen skaalautuvuustestien ja taajuuskaistan käyttöä tarkastelevien testien perusteella.

Avainsanat: Internet of Things, CoAP, LWM2M, IPSO objects

# Preface

# Contents

**Abstract** ii

**Abstract (in Finnish)** iii

**Preface** iv

**Contents** v

**Abbreviations** vii

**List of Figures** viii

**List of Tables** ix

**1 Introduction** 1

1.1 Objectives . . . 2

1.2 Structure of Thesis . . . 2

**2 Background** 3

2.1 Internet of Things . . . 4

2.2 The Web . . . 6

2.2.1 REST . . . 7

2.2.2 HTTP . . . 9

2.3 Communication Protocols for IoT . . . 11

2.3.1 CoAP . . . 11

2.3.2 LWM2M . . . 16

2.4 Virtualization . . . 21

2.4.1 Cloud of Things . . . 21

2.4.2 Docker . . . 22

2.4.3 Greenhouse . . . 23

**3 Design** **25**

3.1 Requirements . . . 25

3.2 Use Cases . . . 26

3.2.1 Simple IoT Devices . . . 27

3.2.2 Fuel Injection System . . . 30

3.2.3 Architecture . . . 32

**4 Implementation** **34**

4.1 Software & Hardware . . . 34

4.1.1 Coap-Node . . . 35

4.1.2 Leshan . . . 37

4.1.3 Hardware . . . 39

4.2 The Implementation of IoT Devices . . . 39

4.2.1 Object Initialization . . . 39

4.2.2 The Communication Logic . . . 40

4.2.3 Virtualization . . . 43

**5 Evaluation** **47**

5.1 Test Environment . . . 47

5.2 Evaluation of Functionalities . . . 47

5.2.1 Fulfillment of Functional Requirements . . . 48

5.2.2 Remaining Issues . . . 49

5.3 Evaluation of Performance . . . 50

5.3.1 Scalability . . . 50

5.3.2 Traffic Analysis . . . 53

**6 Conclusions** **57**

**References** **60**

# Abbreviations

| | |
|---|---|
| IoT | Internet of Things |
| M2M | Machine-to-Machine |
| IP | Internet Protocol |
| CoAP | Constrained Application Protocol |
| WWW | World Wide Web |
| LWM2M | Lightweight Machine-to-Machine |
| API | Application Programming Interface |
| ITU | International Telecommunication Union |
| REST | Representational State Transfer |
| LLN | Low Power and Lossy Networks |
| 6LowPAN | IPv6 over Low power Wireless Personal Area Networks |
| IETF | Internet Engineering Task Force |
| HTTP | Hypertext Transfer Protocol |
| TCP | Transmission Control Protocol |
| UDP | User Datagram Protocol |
| URI | Uniform Resource Identifier |
| AJAX | Asynchronous JavaScript And XML |
| DTLS | Datagram Transport Layer Security |
| CoRE | Constrained RESTful Environments |
| OMA | Open Mobile Alliance |
| ACL | Access Control List |
| ECM | Engine Control Module |

# List of Figures

2.1 IoT stack. . . . 6
2.2 HTTP header. . . . 10
2.3 Example of HTTP request and response. . . . 11
2.4 CoAP message format. . . . 13
2.5 Examples of CoAP CON and NON messages. . . . 13
2.6 The architecture of LWM2M. . . . 16
2.7 The object model of LWM2M. . . . 19
2.8 The architecture of full virtual machine (left) and Docker container (right) . . . 23
3.1 Fuel injection system. [44] . . . 31
3.2 The composite IPSO object with linked objects. . . . 32
3.3 Architecture of the first use case. . . . 33
3.4 Architecture of the second use case. . . . 33
4.1 User Interface of Leshan demo server. . . . 38
4.2 Networking of Docker. . . . 45
5.1 Graph of load average of the system. . . . 52
5.2 The current bandwidth usage of docker0 from iftop. . . . 54
5.3 Traffic load from WireShark with 10 devices. . . . 55
5.4 Packet lengths measured with Wireshark. . . . 56
5.5 Protocol hierarchy . . . 56

# List of Tables

2.1 Data Elements . . . . . 9
2.2 CoAP Response Codes . . . . . 14
2.3 Mapping between LWM2M and CoAP operations . . . . . 18
3.1 Design Requirements . . . . . 27
3.2 IPSO Temperature . . . . . 29
3.3 IPSO objects & corresponding real sensors . . . . . 32
4.1 Reference values for ECM . . . . . 42
5.1 Fulfillment of Functional Requirements . . . . . 48
5.2 Scalability Results . . . . . 51
5.3 New Scalability Results . . . . . 53
5.4 Peak Bandwidth usage of docker0 interface. . . . . 54

# 1 Introduction

While the term Internet of Things (IoT) is relatively new, the concept is not; machine-to-machine (M2M) solutions have been around for few decades enabling devices, such as sensors and actuators, to be connected to one another and communicate on closed purpose-built networks. The IoT, however, is broader concept referring to the use of general Internet Protocol (IP) based technologies and standards, which allow devices to connect directly to the Internet. In this vision *things*, that is, devices in the physical environment become smart enough to be able to communicate and process the data independently, without human intervention. Moreover, they can be remotely managed and monitored, which is another important aspect of IoT.

Such connected devices typically have limited computing power, memory and battery life, which sets technical limitations to the IoT implementations; they need to be as lightweight as possible and the data transmission need to be as minimal and compact as possible. For these reasons, for example Constrained Application Protocol (CoAP) was designed. It is an application layer solution enabling devices to communicate over the Internet. It follows simple design principles, well-known from World Wide Web (WWW or just web), and defines a compact message structure making it suitable for constrained environments. For device management, Lightweight Machine-to-Machine (LWM2M) protocol provides means to monitor multiple constrained devices via lightweight interfaces. It builds upon CoAP and enables interoperability between sensors or actuators and software applications by using a standard data model, LWM2M objects. Based on this model, IPSO Alliance has further defined a set of general purpose objects, which represent common type sensors and actuators.

As the number of IoT devices is expected to grow to billions, also the scalability of the technical solutions used in devices need to be considered. However, setting up physical testbeds and devices may be cumbersome and time-consuming, when it comes to testing IoT scenarios with hundreds, or even thousands of devices. For these reasons, testbeds with virtual devices could be useful. This thesis presents an emulation of IoT devices, which is appropriate for testing purposes. *Emulation* here refers to creating a system that behaves like real physical devices using the exact same protocol stack underneath. The virtual devices communicate using *simulated* data, which is either randomly generated within certain limits or generated using

public data sources. Also in this context, an IoT device means a device, which consists of one or more sensors and actuators.

## 1.1 Objectives

In the beginning of this thesis, there wasn't any sufficient IoT test devices with logic available at Ericsson and existing testbeds were rather cumbersome in terms of configuring and setting them up. The objectives of this thesis is to provide general understanding of IoT and create a testbed of emulated devices using a full CoAP stack with LWM2M and IPSO objects on top of it. The emulator should simplify and ease up IoT related testing, provide devices with logic and enable scaling the amount of devices.

These devices are managed by LWM2M server, which interact with the devices through LWM2M interfaces. IPSO objects are used to represent the common sensors and actuators and organize the data into set of resources that can be read, written or executed. The implementation of the emulator will follow predefined design requirements and create device logic based on two use cases: emulation of several simple IoT devices and emulation of a single complex system.

## 1.2 Structure of Thesis

This Master Thesis consists of 6 chapters. The first chapter is an introduction to the thesis, providing an overview and objectives of the work. Second chapter provides background information and theory relevant to the topic, describing common design principles, the most important protocols for IoT as well as the concept of virtualization. In the third chapter, the design of the practical implementation is presented including requirements for the work and use cases. Next chapter deals with the actual implementation. It presents the relevant programming libraries, device logic and the tools used, such as Docker. The fifth chapter is about evaluation of the work and analyzing the performance of the emulator based on several tests. Finally, conclusions summarize the thesis presenting the most important findings.

# 2 Background

Internet of Things (IoT) and Machine-to-Machine (M2M) communication form the basis of the "future Internet", where small devices with limited processing power, memory and battery, often called smart devices or smart objects, identify and control devices in the physical realm over the internet. [1] [2] The current trend of "anything that benefits from being connected will be connected" reflects well the fast development around the concept, but the realization of IoT is still in the making, both in technology and business perspective.

In general, Internet of Things can be approached with three different views: Internet-oriented (middleware), things oriented (sensors and actuators) and semantic oriented (knowledge).While the Internet-oriented approach looks for the means to integrate the heterogeneous devices and support interoperability within the applications, the starting point of the semantic view is on the services enabled by the new communication technologies. This thesis work focuses on the things oriented view, which emphasizes the active role of smart objects and particularly, the interaction and management of them. Above all, IoT is realized in the application domain, where these different approaches intersect. [3]

IoT architecture is primarily characterized by openness, multipurpose and end-to-end interoperability contributing the use of open standards and development of new lightweight protocols, especially in the application layer [1]. These factors are shifting the M2M industry from highly tailored vertical domain to more abstract horizontal domain, meaning that the customized, vendor-specific solutions aren't valid anymore. It is common that different services have specific requirements and needs, which favors the development of customized solutions. However, such systems usually have inconsistent communications layers and Application Programming Interfaces (APIs), which don't support cross-application integration. Creating a horizontal domain means sharing the common infrastructure and network elements, which brings new challenges for IoT. As vertical application requirements will still exist, support for the horizontal links requires the connectivity and information flow to be invisible to all applications. [4] [5]

This chapter provides background information required to understand the basic

concepts of IoT and virtualization. IoT heavily resides on the fundamentals of web, which are therefore explained in the beginning of this chapter. Next, the most important communication protocols for IoT, CoAP and LWM2M are introduced. Finally the concept Cloud of Things is explained along with the different virtualization tools, such as Docker and Greenhouse that are also used in the implementation.

## 2.1 Internet of Things

International Telecommunication Union (ITU) defines IoT as *"a global infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual) things based on existing and evolving interoperable information and communication technologies"* [6]. It highlights both, IoT's commercial and technological aspects. In technological point of view things or smart objects, such as embedded sensor devices, communicate and share data over wired or wireless IP-based networks, Internet. Especially, IPv6 has made it possible to feasibly connect myriads of devices by providing a large address space and features, such as IP mobility, which were not part of the original design of IPv4 [7]. In addition, IoT extends things into existing World Wide Web, adopting the well-known concepts of web and RESTful design principles. Smart objects typically have constraints on energy, bandwidth, memory and computing power, which has also motivated the development of new lightweight and compact protocols and standards, such as CoAP and LWM2M. In commercial point of view, IoT enables new kind of services by utilizing the mass of generated data. For example, remote management and control of devices offer possibilities in maintenance business.

The decreasing costs and energy consumption of semiconductor components, increasing computing capabilities of sensors and actuators as well as the rise of cloud computing, and widespread adoption of Internet Protocol (IP) have driven the technological development of IoT [1][9]. Moreover, this development has led to a change of traffic patterns; IoT devices exchange near real-time data in numerous, compact messages instead of bulk data. [10]. The new networks consisting of devices with limited resources are called Low Power and Lossy Networks (LLN). For example 6LowPAN [11] introduced by IETF is a low cost communication network, which makes use of small packet size and low bandwidth and hence, allow wireless connectivity in applications with limited power.

Though IoT provides a lot of possibilities, its applicability is ultimately dependent on the service requirements and the capabilities of the devices. For example, security

requirements, low battery time or low processing power of the devices may set restrictions to IoT applications. Thus, for different use cases, it may be useful to classify constrained devices based on their capabilities and identify IoT workflow to understand better the possible challenges. For example, in RFC7228, such devices have been divided into three classes; class 0 devices are the most constrained, roughly having much less than 10 KiB RAM and less than 100 KiB Flash, while class 2 devices are the least constrained having about 50 KiB RAM and 250 KiB Flash. This leaves class 1 devices in between. [2] Regardless of the device class, a simple IoT work flow of a device in general can be described with three stages [8]:

1. **Sensing**: smart devices collect data from the environment, for example about temperature.
2. **Action**: smart devices process the received data and trigger an action.
3. **Feedback**: smart devices provide feedback of the current status or action to the management server or other devices.

For class 0 devices, actions, such as updating configurations or storing the security information need to be very simple and minimal, whereas more capable devices can handle larger data packets and maintain larger database. In terms of security, it is relevant to ask, who can access the management server, does the data need to be encrypted or what kind of actions are allowed for devices to trigger. Reliability requirements also effect on the protocol choices and depend on the device capabilities.

### IoT Stack

The IoT stack used in this thesis is illustrated in figure 2.1. It consists of standardized protocols, APIs, data models and the sensor application. Network connectivity between devices is provided by common physical and data link layer protocols, for example Ethernet, Wifi or 802.15.4, which is especially intended for wireless network of low-power devices. IoT utilizes the ubiquitous IP protocols, IPv4 and IPv6, for routing along with the 6LowPan, created by IETF for constrained environments. [12][13]

The scalability of web has promoted RESTful design of IoT applications suggesting the use of widely deployed Hypertext Transfer Protocol (HTTP). However, HTTP over Transmission Control Protocol (TCP) is too heavy to use with the power and memory limited devices in most cases and thus, IETF has developed a more compact protocol CoAP, which communicates over User Datagram Protocol (UDP) by default.

[13] LWM2M protocol together with IPSO object model provides a lightweight, universal interface for IoT architecture. The application software running on sensors and actuators relies on the full-stack. [12]

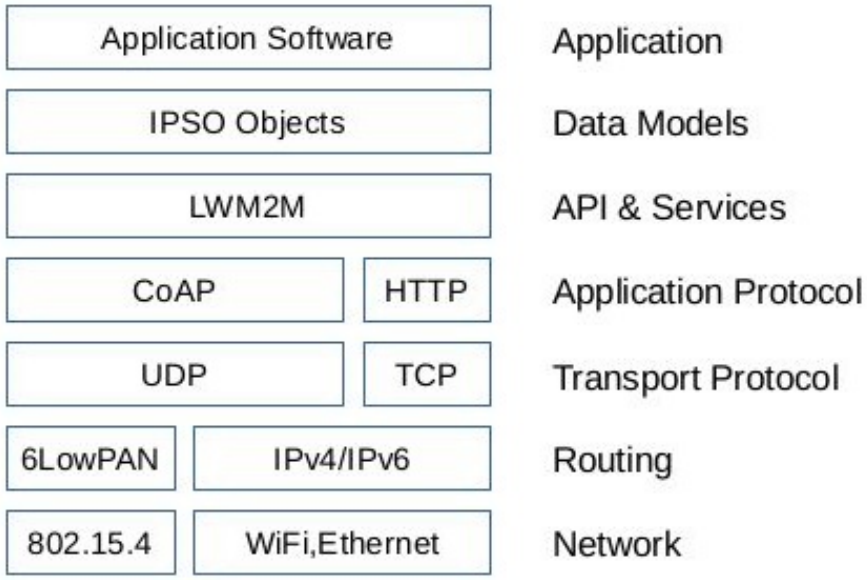



**Figure 2.1: IoT stack.**

## 2.2 The Web

One approach to realize IoT is to reuse the existing web technologies and standards and extend them to cover the network of constrained devices [14]. This Web of Things -approach facilitates the integration of devices with the Web considerably, since mapping traffic between the common web and emerging IoT protocols doesn't require any complex procedures making the development easier and faster.

Web focuses on resources and content, and uses URI-based addressing scheme [15]. The resource-oriented view seems appropriate also in constrained IoT environments, where organizing and handling resources require careful planning and optimizing. Adopting the similar media types and addressing format further unifies the web and IoT. To understand better the emerging IoT technologies and their benefits, it is important to take a look first at the key concepts of web, namely RESTful design and HTTP, which are explained in the next sections in more detail.

### 2.2.1 REST

Fielding [16] defines Representational State Transfer (REST) as a architectural style, which provides design principles for distributed hypermedia systems. It introduces a set of constraints that aim for scalability, component independence and efficiency. In particular, the REST architecture is based on the client-server model, which separates the tasks of issuing requests (client) and providing services (server). Having two separate roles simplifies the implementation of components, but also improves portability, since components can be developed independently.

Fielding used REST to design HTTP/1.1 and URIs, making it one of the cornerstones of web. Hence, it has been a natural step to adopt the RESTful architecture for IoT as well, which has resulted in designing CoAP, a mere redesign of HTTP. Nevertheless, while the principles of REST has remained the same, CoAP has taken things further and extended the HTTP semantics. In HTTP, the endpoints have very distinct tasks: a client, for example a web browser, sends requests, while a server, for example a computer hosting a web site, provides resources accordingly. In CoAP on the other hand, the endpoints operate in both roles; for example smart devices act as a client when registering with a resource directory, but fulfill a server role when providing sensor data [17]. Moreover, CoAP has changed the weight of the roles. HTTP server carries greater role in processing, storing and delivering the data than HTTP client, but CoAP turns the roles around. The CoAP requests are more complex in a sense that client often needs to process data of its own resource database, whereas the request handling by CoAP server is simple; resources are easily accessed from device's own database with predefined methods.

#### Design Constraints

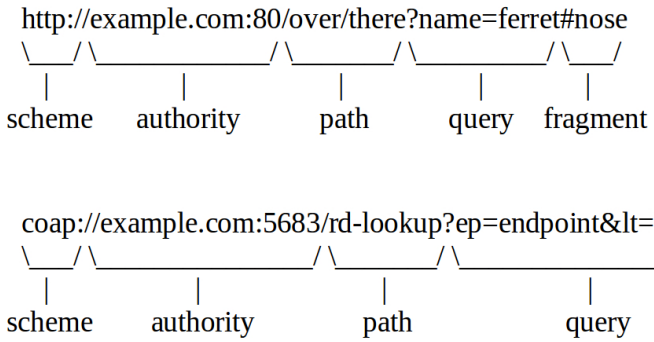
One of the fundamentals of REST architecture is the stateless nature of communication between the components, which removes the need for an awareness of the overall component topology. That is, the session state is kept mostly on the client. Each request must be self-descriptive, containing all information needed to understand the request. In addition to the improved visibility of the system, the stateless constraint increases scalability, since server won't occupy resources for long. As a trade-off, the repetitive data increases, decreasing the network performance. [16] IoT systems benefit from the stateless constraint, since it lowers memory requirements of servers, which typically are constrained and need to handle large amount of clients [17].

REST also induces a cache constraint and a layered design, where independent,

hierarchical layers restrict the knowledge of each component to a single layer. In other words, a client cannot see beyond the server it is connected to, which simplifies the overall architecture. Focus is also on uniform interface between the components, which is optimal for common cases of web and encourages independent development of services.[16] This type of design is suitable for IoT systems; not only is it simple, but also already well-known from the use with HTTP. Hence, the integration of web services and IoT becomes easier.

### Data Elements

In relation to hypermedia systems, one of the most important aspects of REST is the concept of data elements (table 2.1). They form a common base for RESTful application design and provide mutual language between the client and server, which promotes interoperability. At its core, the information is organized as set of resources, which can be retrieved, created, deleted or modified and each resource is identified, referenced and accessed through a global unique resource identifier (URI). HTTP and CoAP URIs have the common structure, which is shown below with the definitions of the components [18][19].




- **Scheme** creates a namespace for resources. For example “http” or “coap”.
- **Authority** identifies an entity, such as the server "www.example.org" in the current schemes. It is an IP address or a host name, with which the server can be reached, followed by an optional transport layer port number. In case of coap scheme, the default port 5683 is assumed if no port is given.
- **Path** identifies a resource within the scope of URI’s host and port. Path segments are separated by a slash (“/”) character. For example “/over/there” or “/rd-lookup”.

- **Query** contains a sequence of arguments to parameterize the resource further. The component begins with the question mark ("?") character and the arguments are separated by the ampersand ("&") character. For example "?ep=endpoint&lt=lifetime".

- **Fragment** refers to some part of the resource, such as a section in an HTML document. It is separated by the hash ("#") character. For example "#nose". It is relevant for HTTP, but not for CoAP.

Components of the network exchange representations of the resources, such as HTML documents or JSON templates, in order to be able to interpret the state of the resources and perform actions on them. This introduces yet another important data element for hypertext-driven communications, a media type. Media type describes the data format of representation, that is, a set of rules for data encoding/decoding. It is sent in the "Accept" or "Content-Format/Content-Type" options field of the HTTP or CoAP header. As HTML is a typical media type for HTTP, for example JSON or TLV is typical for CoAP. The structure of the representation is specified by the representation format, which can be link or form. [21] Links are typically used in navigating among resources, while forms are used to change a resource state. For example CoAP resource discovery requests are sent in link-format with target attributes and more specific read requests are sent in form-format with GET method and a description of Content-Format. Finally, components need to define the meaning of the received requests and responses and process them by using control data.[16]

**Table 2.1: Data Elements**

| Data Element | Examples |
|---|---|
| Resource | Target of hypertext reference |
| Resource identifier | URL, URN |
| Representation | HTML document, JPEG image |
| Representation metadata | Media type, Representation format |
| Resource metadata | Source link |
| Control data | Cache-control |

### 2.2.2 HTTP

The Hypertext Transfer Protocol (HTTP) [24] is a widely deployed stateless, application layer protocol for distributed hypermedia systems, which has been used for

web transfer since 1990. It follows a REST based request/response model building on resources, URIs, media types and methods, which make different kind of requests and functionalities possible. They include for example retrieval, search and editing of resources.

HTTP provides reliable data transfer, which is usually implemented over TCP/IP connections, but other protocols may also be used. The default TCP port is 80. HTTP communication takes place between the user agent (client) and a server, which hosts the requested resource, but different intermediates, such as proxies or gateways may be involved. HTTP connection is persistent by default meaning that only one TCP connection is established and used to carry out requests and responses. Connection can be secured by using encryption provided by TLS protocol [25] or its predecessor SSL [26].

### Messaging Model

There are two types of messages, requests and responses, to transfer entities, that is payload. In general they consist of start-line, message headers, empty line indicating the end of headers and possible message-body that carries entity-body. The headers in turn consist of general-header, request-header, response-header and entity-header fields. The message structure is shown in figure 2.2. General header fields are relevant for both requests and responses. For example "Connection" field is used to signal the closing of TCP connection, which by default would otherwise remain open. Entity header fields, such as "Content-Type" provide information about the present entity-body.[24]

| | |
|---|---|
| Start-Line | Request-Line \| Status-Line |
| Message Headers | General-Header |
| | Request-Header \| Response-Header |
| | Entity-Header |
| Message-Body | Entity-Body |

**Figure 2.2: HTTP header.**

As client sends a request to the server, the message always includes the method to be applied on the target resource, the identification of the target resource (URI) and the protocol version within the start-line called a Request-Line. In addition, a request message may contain header fields, which provide additional information

about the request or client. For example, “Host” and “Accept” fields can be used to specify the target host and acceptable data format. Server responds with a message, which starts with a Status-Line containing the protocol version, status code and its textual presentation. Also response message may have header fields that provide additional information about the server or the requested resource, such as ’Location” to redirect request to other location than the one indicated in the request-URI. An example of HTTP request and response is shown in figure 2.3.[24]


**Figure 2.3: Example of HTTP request and response.**

## 2.3 Communication Protocols for IoT

IoT devices may communicate over web using for example HTTP, but the constraint environment sets limitations. That is why CoAP was developed. It is basically a redesign of HTTP, but in a more appropriate form for constrained devices. CoAP uses lighter transfer mechanisms and has a shrinked header with less options. It also provides M2M-specific features, such as resource observation, which is relevant in IoT scenarios, where devices need to have up-to-date knowledge on the latest value of a specific resource. In this thesis CoAP is used on the emulation of IoT devices, LWM2M protocol on top of it. LWM2M provides a management and communication interface and a data model for smart objects. Its several APIs are used to define for example device, location, connectivity and server information. In addition, IPSO Alliance has defined a model for a set of smart objects, which can be used to represent the actual sensors and actuators in a device.

### 2.3.1 CoAP

The Constrained Application Protocol (CoAP), defined in [19], is a lightweight RESTful web transfer protocol designed for constrained devices and networks, especially M2M applications. CoAP provides basic concepts of web, such as URIs and Internet

media types, while defines specialized features for M2M, for example low overhead, multicast support and built-in resource discovery.

CoAP utilizes a request/response model between application endpoints like HTTP, but as a distinction CoAP messages are exchanged asynchronously between the endpoints over UDP by default. HTTP is a synchronous protocol in a sense that client issuing a request waits for the corresponding response, before continuing execution of other tasks. CoAP on the other hand can handle multiple simultaneous requests independently, thus performing more efficiently. In web though, the use of AJAX, a set of web development techniques, has enabled asynchronous HTTP requests [20].

UDP provides more lightweight data transfer than for example connection oriented TCP having no built-in reliability mechanism. TCP provides reliability and handles ordering of messages and tracking connections, which makes it more heavyweight. Nevertheless, recently there has been work in progress about using CoAP over TCP [27] in environments, where for example UDP based protocols may not be well received. Also in less constrained environments, where is only a limited amount of devices, TCP connections can easily be handled despite the possible decrease of efficiency. In fact, in near future TCP may be as relevant solution as UDP. In addition to the current, mandatory UDP binding, CoAP supports SMS option as well. [19] [23]

CoAP also provides few security modes: no security, key-based (PreSharedKey & RawPublicKey) and certificate-based security. In secure mode, Datagram Transport Layer Security (DTLS) [28] over UDP is used to secure the end-to-end communication channel. [19] [23]

### Messaging Model

CoAP requests and responses are carried in binary format messages (figure 2.4). CoAP message format starts with a compact, fixed-size 4-byte header, which consists of CoAP version number (2-bit), message type indicator (2-bit), token length (4-bit), message code (8-bit) and message ID (16-bit). The header is followed by a variable-length token value, which is used to match requests to responses, and further by optional sequence of options and payload marker indicating the end of options and a start of payload that takes up the rest of the datagram. [19]

There are four types of CoAP messages: Confirmable (CON), Non-Confimable (NON), Acknowledgement (ACK) and Reset (RST). Reliability is provided by marking a

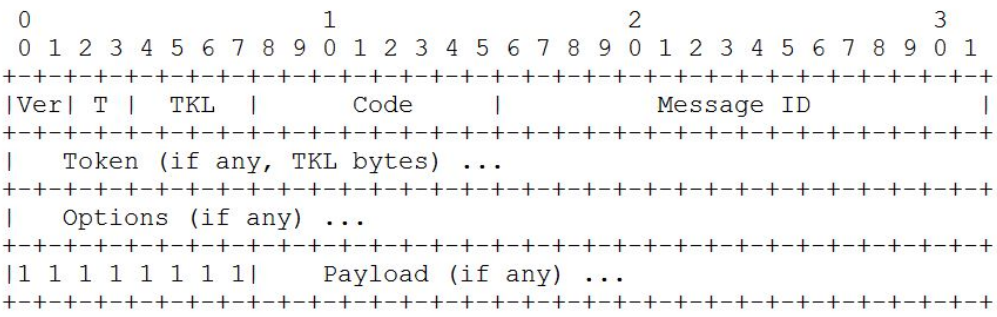



**Figure 2.4: CoAP message format.**

message as Confirmable. CON messages are retransmitted at exponentially increasing time intervals (back-off) until the recipient responds with acknowledgement message with the same message ID (for duplicate detection) or timeout is reached. The payload of the response is typically piggypacked in ACK message, but ACK may be empty if response is not immediately available. In this case, a separate response is sent later. If recipient cannot process CON message, it sends Reset message instead of ACK. On the contrary, a message that doesn't require reliability is marked as Non-confirmable. NON message is sent with a message ID, but not acknowledged. Instead, the response is sent in a new NON message. Example of Client/Server communication is shown in figure 2.5. [19]

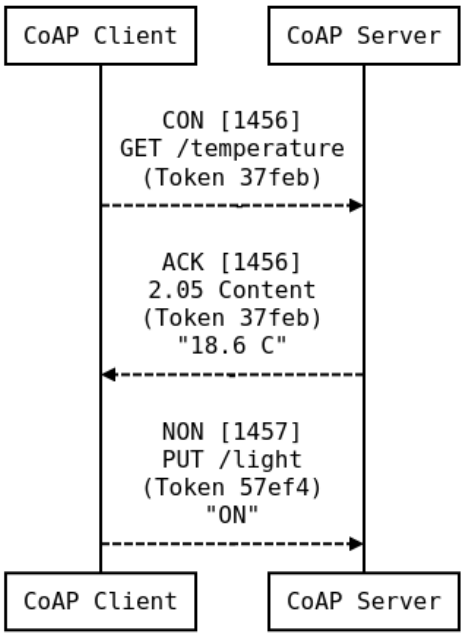



**Figure 2.5: Examples of CoAP CON and NON messages.**

### Methods & Response Codes

CoAP requests and responses are carried in CoAP messages, which include either a Method Code or a Response Code, respectively. CoAP makes use of GET, PUT, POST, and DELETE methods in a similar manner to HTTP. The GET method retrieves a representation for the information that currently corresponds to the request URI. The POST method requests that the representation enclosed in the request is processed. Depending on the target resource, POST request usually results in a new resource being created or the target resource being updated. The PUT method requests that the resource in question is updated or created with the enclosed representation. The representation data format is specified by the media type and the possible content coding. The DELETE method requests that the resource identified by the request URI is deleted. [19]

Also most of the Response Codes resemble the Status Codes of HTTP with few exceptions. The numbering is slightly modified though, for example HTTP 404 “Not Found” is written as 4.04 in CoAP. [24] [19] CoAP supports three classes of response codes defined in table 2.2.

**Table 2.2: CoAP Response Codes**

| Code | Description |
|---|---|
| 2.xx Success | This class indicates that the request was successfully received, understood and accepted. |
| 4.xx Client Error | This class is intended for cases in which the client seems to have erred. |
| 5.xx Server Error | This class indicates to cases in which the server is aware that it has erred or is incapable of performing the request |

### Resource Discovery

CoAP uses the "coap" and "coaps" URI schemes for identifying CoAP resources and providing means of locating the resource. One important feature of CoAP is the resource discovery, which provides information of all public resources offered by endpoints. This is particularly useful in M2M environments, where devices need to communicate with each other without human-intervention. For this purpose CoAP endpoints support the Constrained RESTful Environments (CoRE) Link Format of discoverable resources, described in [29]. The resource discovery provided by HTTP is generally called "Web Discovery", while description of relations between resources

is called "Web Linking". CoRE Link Format extends the “Web Linking” with specific M2M attributes including for example resource type “rt” attribute and interface description “if” attribute. The former one is a application specific description of a resource, for example “outdoor-temperature” or “kitchen-light”. The latter one describes the interface used to interact with the resource. It may be an URI, URN or a name, for example “sensor”.

CoAP employs an entity called Resource Directory (RD), which hosts the description of resources and a well-known resource path as a default entry point for the RD. Performing a GET request to the the well-known interface returns all available links in CoRE Link Format. The structure of a CoAP URI for resource discovery is:

coap://example.com:5683/.well-known/core

In addition, this interface supports filtering on link format attributes using query strings. For example a request GET /.well-known/core?rt=outdoor-temperature would return links that match that particular resource type. Nevertheless, server is not required to support filtering. Resource discovery can be performed either with unicast (one-to-one) or multicast (one-to-many) transmission.

### Observation

The state of a resource may change over time, which can be undesirable for a client that is interested in the current representation of the resource. This has motivated the development of mechanism for a CoAP client to observe a resource on CoAP server [30]. According to specification client can request a representation of a resource be updated by the server periodically until client is no more interested in it.

The minimum and maximum time between the coming updates, notifications, are configurable and choosing the values is an important design matter in IoT applications. For example, for monitoring of public waste management system, one notification per hour may be enough [31], while monitoring presence in a room should be frequent, if the change in state is supposed to trigger an immediate action. The time interval has a significant effect on traffic load in the network.

This observer design pattern consists of two components called subject, which is a resource in the CoAP namespace, and observer, which is a CoAP client interested in the current representation of the resource. Observer registers its interest by sending an extended GET request to the server. When the state of resource changes, server sends notification, that is, an additional CoAP response with the updated representation of the resource to the clients in the list of observers.

### 2.3.2 LWM2M

LWM2M is a an open industry standard, which provides a compact communication interface and a data model enabling remote device management and service enablement of embedded IoT devices and resources [12]. It defines APIs for example for security, connectivity monitoring, firmware upgrade and software management. Open Mobile Alliance (OMA) designed the application layer protocol especially for constrained environments; it utilizes CoAP as an underlaying communication protocol and simple object model for organizing data. Moreover, the RESTful design of LWM2M is appealing to software developers, which makes LWM2M a potential solution for emerging IoT systems [23].

The architecture of LWM2M consists of three components: LWM2M client, LWM2M server and LWM2M bootstrap server. In essence, LWM2M client is a client software running on a M2M/IoT device, and LWM2M server is a server software on a M2M/IoT management and service platform. [23] Typically, the LWM2M server is located in data center and hosted by for example M2M Service Provider or Network Service Provider [12]. It is important to acknowledge that both, LWM2M client and server have the roles of CoAP client and server. The LWM2M architecture without bootstrap server is shown in figure 2.6.

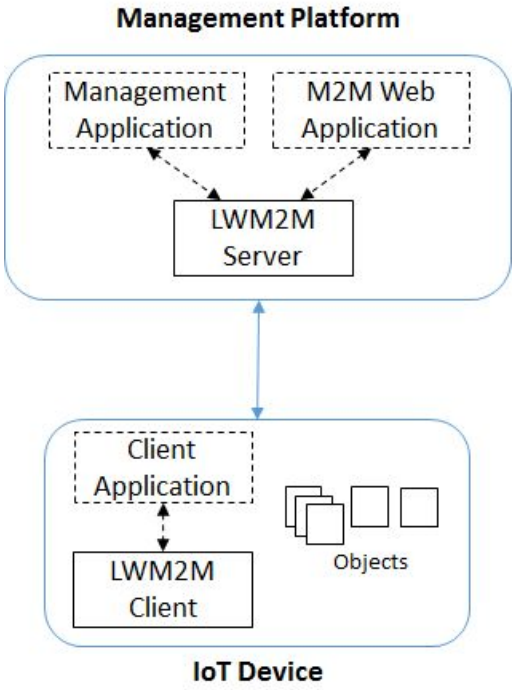



**Figure 2.6: The architecture of LWM2M.**

### Interfaces

The communication mechanism between the components relies on four logical interfaces: 1) Bootstrap, 2) Device Discovery & Registration, 3) Device management & Service Enablement and 4) Information Reporting. Table 2.3 shows the mapping between all the LWM2M operations and CoAP operations.

Bootstrap Interface is used to enable the connection between LWM2M client and LWM2M server(s). The provisioned information between the components can be categorized into two types: LWM2M Server Bootstrap Information and LWM2M Bootstrap Server Bootstrap Information. Bootstrapping may be performed in four different modes: Factory Bootstrap, Bootstrap from Smartcard, Client Initiated or Server Initiated Bootstrap. [32]

Registration Interface is used by the LWM2M client to register with LWM2M server(s), maintain registration and deregister from LWM2M server. During registration, client sends "Register" operation, which provides server the information required to contact the client and maintain the session. For example, each client must provide a unique client name, which is stored by the server. Also the list of client supported objects and existing object instances is forwarded. After registration, the client performs either periodically or on request, an "Update" operation to update its registration information. Finally when client wants to discontinue to use the services of LWM2M server, "Deregister" operation is triggered. [32]

Device management and service enablement Interface is used by the LWM2M server to access and act upon LWM2M client object instances and resources. The possible operations include "Create", "Read", "Write", "Execute", "Delete", "Write Attributes" and "Discover". These operations allow server to perform several actions on the resources, such as create, retrieve, update or configure resources.[32]

Information Reporting Interface is used by the LWM2M server to observe changes on a resource of LWM2M client and receive notifications of new values. This is implemented by sending "Observe" request to the LWM2M client, after which LWM2M client performs "Notify" operation to inform the server of the new resource values. Observation ends, when server sends "Cancel Observation" request. [32].

**Table 2.3: Mapping between LWM2M and CoAP operations**

| LWM2M Operation | CoAP Operation |
|---|---|
| Request Bootstrap | `req: POST /bs?ep=ClientName`<br>`rsp: 2.04 Changed` |
| Register | `req POST /rd?ep=ClientName&lt=8600&lwm2m=1&b=u`<br>`rsp: 2.01 Created` |
| Update | `req: PUT /rd/AkTr0mE9v6?lt=12000&b=u`<br>`rsp: 2.04 Changed` |
| Deregister | `req: DELETE /rd/AkTr0mE9v6`<br>`rsp: 2.02 Deleted` |
| Read | `req: GET /3303/0/5700`<br>`rsp: 2.05 Content` |
| Discover | `req: GET /3303/0/5700 Accept: application/link-format`<br>`rsp: 2.05 Content` |
| Write | `req: PUT /3303/0/5700`<br>`rsp: 2.04 Changed` |
| Write Attributes | `req: PUT /1/0/0?pmin=10&pmax=60&lt=12000`<br>`rsp: 2.04 Changed` |
| Execute | `req: POST /3/0/1`<br>`rsp: 2.04 Changed` |
| Create | `Req: POST /3303/1`<br>`rsp: 2.01 Created` |
| Delete | `Req: DELETE /3303/0/5700`<br>`rsp: 2.02 Deleted` |
| Observation | `req: GET /3303/0/5700 Observe: 0`<br>`rsp 2.05 Observe: 1` |
| Cancel Observation | `Reset message`<br>`req: GET /3303/0/5700 Observe: 1`<br>`rsp: 2.05 Content` |
| Notify | `rsp: 2.04 Changed Observe: 2`<br>`rsp: 2.04 Changed Observe: 3 ...` |

### Object Model

LWM2M introduces a simple object model, where available information on LWM2M client is constructed as resources, and resources are organized logically into objects. Moreover, objects or resources may have multiple instances. The structure is illustrated in figure 2.7. Each object and resources within are addressed using URIs,

which consist of unique identifiers, for example:

- “3/0/1”: identifies the “model number” resource of the "device" object
- “3/0”: identifies the whole “device” object instance

Objects and resources must support at least one operation (read/write/execute/create/delete). In addition, LWM2M provides an access control mechanism per object instance allowing different servers to have different access rights upon objects and resources. This is realized by using Access Control Lists (ACLs) on access control object identifiers. [32]

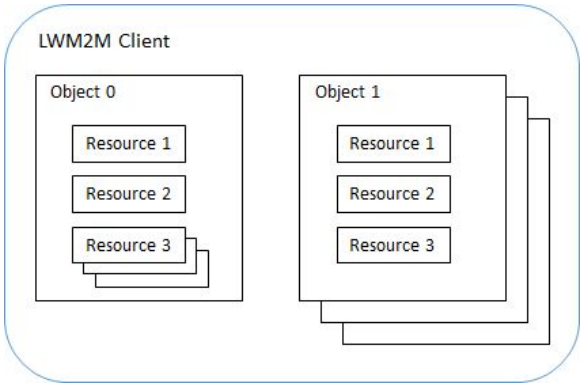



**Figure 2.7: The object model of LWM2M.**

LWM2M defines an initial set of management objects, which are defined below [33]:

- **Security**: handles the security data between the LWM2M server and the client. One object instance should contain information about the bootstrap server.
- **Server**: provides the data and functions related to the management server.
- **Access Control**: defines the access rights of LWM2M server. Access control object instances contain ACLs that determine the allowed operations on a given object instance.
- **Device**: LWM2M server can query the device specific information via this object. It also provides device reboot and factory reset operations.
- **Firmware**: defines resources, which are needed for firmware updates. It enables for example installation of firmware packages.
- **Location**: provides information about the current location of the device, such as latitude and longitude.

- **Connectivity monitoring**: defines resources, such as "Link Quality" and "Router IP Address", which assist in monitoring the status of a network connection.
- **Connection statistics**: provides statistical information about an existing network connection, for example about transmitted or received data.

### IPSO Objects

IPSO Alliance has defined a set of general purpose smart objects, which represent common sensors and actuators. They ensure high level interoperability on the application layer by exposing a universal interface for devices and servers without the need to change the underlaying application logic. This object model is designed to operate on top of any RESTful protocols, and is based on OMA LWM2M specification. [34] [35] The object model consists of four parts described below [34].

**Object Representation** follows OMA LWM2M object model, where objects and resources are structured into URI path components: Object Type ID/ Object Instance ID/ Resource Type ID. Object Type ID specifies a measurement point, for example a temperature sensor, whereas Object Instance ID represents a single instance of an measurement point. Object may have several resources, that is, properties such as current temperature value, which have their own Resource Type ID. For example 3304/0/5601 represents a humidity sensor (ID 3304) and its maximum measured value (ID 5601).
**Data Types** that the resources can be defined to be include string, integer, float, boolean, opaque, time and objlnk, which are defined in OMA LWM2M specification.
**Operations** supported by IPSO Objects also origin from OMA LWM2M specification. They include Read, Write, Execute, Create, Delete, Set and Discover depending on the target, which can be object, instance, resource or attribute.
**Data Formats** for transferring resource information include Plain Text, CoRE Link Format, Opaque, TLV and JSON. In terms of media types for example text/plain, application/cbor, application/json, application/tlv and application/link-format are supported.

## 2.4 Virtualization

This section discusses virtualization in terms of utilizing the concept in real IoT scenarios and utilizing it in the implementation part of this thesis. As the data generated by IoT devices increses, storing and processing it locally becomes difficult in real life scenarios. Cloud computing provides means to lighten the processing load of constrained devices. For implementing the testbed of virtual devices in this thesis, tools that enable building the whole virtual environment is needed. For example Docker is such tool that can be used to create appropriate runtime environment for the applications. First, the concept of Cloud of Things, which combines IoT and cloud computing, is introduced. Next, tools called Docker and Greenhouse are presented in more detail.

### 2.4.1 Cloud of Things

Ericsson estimated in 2016 that the number of connected devices in total is expected to be around 28 billion by 2021, of which close to 16 billion will be related to IoT [38]. Thus, it is clear that the amount of data IoT generates is going to be vast and storing it locally is not possible anymore. Moreover, the data needs to be processed to a more suitable form for services, which can't be managed in constrained IoT devices. This makes the importance of cloud computing evident. Aaza et al. (2014) refers to integrating IoT with cloud computing as Cloud of Things. [36]

Cloud computing provides virtual pool of computing resources and data to devices over the Internet. It is dynamic environment that manages and allocates resources automatically and process data into more relevant form for different services. [37] User only needs to have a working Internet-connection to be able to utilize such services without any concern about the maintenance. Cloud computing presents different type of services, such as Software as a Service (SaaS) and Platform as a Service (PaaS). SaaS provides user an access to an application on pay-as-you-go basis, whereas PaaS refers to renting a platform, which contains all the required resources needed to build applications. [36]

Though Cloud of Things has a lot of potential benefits enabling enhanced services, it also has several issues related to for example protocol support, energy consumption and security. IoT may utilize different protocols, such as ZigBee or 6LowPan, depending on the environment and sensors used. Interoperability would require support for these protocols on a gateway device. Therefore, one of the suggested

solutions is mapping the standardized protocols on a gateway. [36] In relation to energy issues, the operations of sensors and actuators as well as the connectivity with the cloud may consume a lot of energy, which becomes a problem, when billions of devices are connected to the network. For example new compression techniques and more efficient encoding/decoding methods improve energy efficiency as well as optimizing the use of resources on the cloud side. [36] Moving the IoT applications towards the cloud also brings security concerns due to for example lack in the knowledge of real location of data and trust in service providers. [37] These are the issues that still need to be addressed and solved before fully utilizinbg the Cloud of Things concept.

### 2.4.2 Docker

As the implementation of emulated devices is based on having only virtual components, means for virtualizing them is required. Thus, a tool called Docker is utilized. It is an open platform, which enables distributed applications to run seamlessly regardless of the environment. Docker containers package the application with all the necessary runtime dependencies, system tools and libraries, and isolate it from other applications and the underlying infrastructure. That is, they guarantee that the applications run always the same way. Containers are compatible with all major distributions of Linux and Microsoft Windows and operate on top of any infrastructure; any computer or cloud will do. [39] With Docker, we can create separate containers for each of the emulated devices. This way, each device will have their own system resources and a unique IP address.

In the core of the technology is the Docker Engine, that control containers and images. In essence, image is a layered filesystem, which shares common files, and container is an instance of an image. Containers running on a single host machine share the kernel of the operating system and hence, use less memory (RAM) and start instantly. [40] Docker improves portability and scalability. Applications can be deployed fast, since the environmental and infrastructural constraints can be circumvented easily. In addition, Docker enables running several containers without consuming much space and memory of the host machine. [39]

#### Docker vs. Full Virtual Machine

Full virtual machine, or system VM, provides a complete system environment including the quest operating system along with the application and required binaries and

libraries. It has its own set of system resources allocated and does minimal sharing. VM provides quest OS an access to virtualized hardware resources. While the system isolation is more thorough, full VM is much heavier than Docker. [41]

In contrast to VM, Docker containers share the kernel of the host OS and run as isolated processes in user space. Containers running the same OS image share the size of the image. Thus, docker can easily run hundreds or even thousands of containers simultaneously. The architectural difference is visualized in figure 2.8. All in all, VM provides full isolation and guaranteed resources, while Docker provides less isolation, but lightweight platform for multiple simultaneous containers. [39]

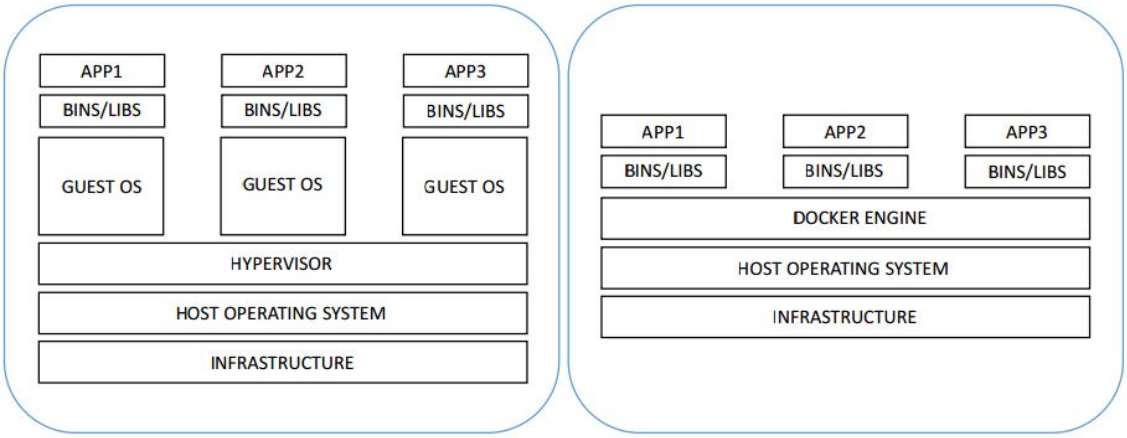



**Figure 2.8: The architecture of full virtual machine (left) and Docker container (right)**

### 2.4.3 Greenhouse

Greenhouse is a DevOps tool suite by Ericsson (currently internal use only), which provides custom functionality to existing platforms, for example configuration management or automatic proxy handling. It is based on service choreography rather than service orchestration [43]. Service choreography is a decentralized form of service composition, which has a global perspective. It tracks message sequences between multiple endpoints instead of actions by a single party. Service orchestration is a centralized model, where a single business process, orchestrator, control the interaction of multiple services. [42]

The Greenhouse architecture consists of four components: Bitverse, Facehugger, Farmer and Web-UI. Greenhouse creates decentralized peer-to-peer (p2p) network of

containers, to which it injects its management processes to manage the application. The architectural components are discussed briefly below [43].

- **Facebugger**: a tool to inject Greenhouse functionality to existing docker container. User chooses, which tools are activated.
- **Farmer**: a gateway to support the different underlying platforms, such as Docker or Apcera. It consists of a daemon process (Farmerdaemon) and an executable (Farmer CLI).
- **Bitwerse**: a Distributed Hash Table (DHT) which handles state synchronization and messaging between distributed components. It implements two kind of nodes: supernodes, which manage routing in bitverse network and edgenodes, which are integrated in clients and support sending messages to other clients.
- **Web-UI**: a web-based user interface that visualize the system.

As mentioned above, Greenhouse functionality is based on a set of tools, which can be independently activated. In this thesis work, the relevant tools are Jekyll and Tesla. Jekyll coordinates the running containers by starting and stopping them as appropriate. Tesla is used to find dependent containers in the network, connecting for example, the LWM2M clients to the LWM2M server. This is done by determining the polarity of the component: positive polarity (+) means the container is a “producer” (eg. LWM2M server) and a negative polarity (-) means the container is a “consumer” (eg. LWM2M client). Tesla automatically resolves mapping between virtual and physical ports and determines the IP address of the host machine, making it possible to run several identical containers on the same machine.

# 3 Design

In this thesis, an IoT device consists of one or more virtual sensors, which harvest information from the simulated environment and convey it to the management server or virtual actuators, which function like a switch being able to change the status of the device and inform the management server about it. The emulation of such devices is implemented by creating a LWM2M client software, which utilizes random data and is able to perform several basic functions through LWM2M and IPSO objects. As example IoT use cases, we present several simple IoT devices communicating with the management server and to each other, and a single complex system communicating with the management server.

This chapter defines the design requirements for the LWM2M clients and presents the use cases in more detail. Also the chosen IPSO objects and the architecture of the use cases are discussed.

## 3.1 Requirements

One of the goals of this thesis is to implement few virtual devices, LWM2M clients, which can be managed by the LWM2M server and communicate directly with each other. There are set of client specific requirements stemmed from the LWM2M specification, but also more general system requirements, all of which are listed in table 3.1 with the indication of their importance in the implementation.

In general, the most important client specific functionalities are LWM2M client's ability to register, read, write and execute resources of single-instance objects and deregister. Single-instance object refers to an object, which is allowed to have only one instance, while multi-instance object may have several different instances. The division between single-instance and multi-instance objects is made here, since they support different data formats; single-instance objects may be dealt with for example plain text, whereas multi-instance objects must support JSON and/or TLV format. As important as communication between LWM2M server and client is, also the direct communication between LWM2M clients is important in real IoT scenarios, where devices need to be able to communicate without a human intervention.

In addition, there are few performance requirements. For example scalability, bandwidth usage and memory usage of the devices (docker containers) are important factors. LWM2M server should be able to handle hundreds of simultaneous devices, which send periodically data to the network. As the number of connected devices increase, the bandwidth and memory usage may become a problem, if not taken into account in the implementation.

## 3.2 Use Cases

The implementation part involves two use cases: emulation of several simple devices and emulation of one complex system consisting of multiple sensors and actuators. In the former case, four different virtual devices, a weather observer, a radiator, a presence detector and a light controller are connected to a network. A weather observer and a radiator together control a room temperature, while a presence detector and a light controller manages the status of the lights depending on the occupancy of the room. This use case is chosen, because it demonstrates simple IoT scenarios showing the basic use of common sensors and actuators. Moreover, it is appropriate and realistic use case for scalability testing. For example buildings usually have several lights and radiators that could be controlled with means of IoT.

The second use case presents a complex system, engine fuel injection system of a car, which consists of multiple virtual sensors and actuators. The heart of every modern car is a engine control module (ECM), a microcontroller, which controls several engine operations for the engine to run efficiently and economically. Its main tasks include control of fuel injection and fuel ignition. The focus of this use case is on the fuel injection. ECM receives information from different sensors, such as throttle position, cramshaft speed, coolant temperature and mass air flow sensors in real time and adjusts the time fuel injectors are open. Fuel injectors are basically ON/OFF switches, which open and close the injector valves accordingly. ECM also needs to check that the fuel is pumped at the correct pressure and make corrections if necessary. This use shows, how a complex systems may be modeled and managed in IoT.

**Table 3.1: Design Requirements**

| Functionality | Description | Importance |
|---|---|---|
| Registration/Deregistration | LWM2M client is able to register/deregister to/from the management server. | High |
| Resource initialization | LWM2M client is able to initialize objects and resources. | High |
| Single-instance support: read/write/execute | LWM2M client is able to read, write and execute resources of single-instance objects. | High |
| Multi-instance support: read/write/execute | LWM2M client is able to read, write and execute resources of multi-instance object and read the whole instance. | Medium |
| Create/Delete | LWM2M client is able to create and delete object instances. | Medium |
| Observe/Notify | LWM2M client supports observe/notify concept. | High |
| JSON/TLV format | LWM2M client supports JSON and/or TLV format required for handling multi-instance objects. | High |
| Device-to-Device communication | LWM2M clients are able to communicate directly to each other. | High |
| Scalability | LWM2M server should be able to handle hundreds of simultaneous clients. | High |
| Bandwidth Usage | LWM2M client should generate small amounts of traffic in nearly real time instead of bulk data. | Medium |
| Packet Size | CoAP messages should fit within a single IP packet fragmentation to avoid fragmentation. | High |

### 3.2.1 Simple IoT Devices

This use case includes four devices: a weather observer, a radiator, a presence detector and a light controller. A weather observer simply provides temperature readings utilizing either random data or an OpenWeatherMap data source to get current

weather data from real weather stations. It has a programmable target temperature value for the room to which it compares the current temperature and if necessary, sends a control message to a radiator. A radiator switches its status to ON or OFF based on the control message. A presence detector observes occupancy of a room and sends a control message to a light controller, when its status is changed. A light controller turns the lights ON or OFF based on the control message.

For the emulation of these simple devices, IPSO temperature, humidity, light control, presence, set point and general actuator objects were chosen. They represent very common type of sensors and actuators, which enable remote data measurement (e.g. temperature sensor) and control (e.g. light control actuator) and thus, are appropriate targets for emulation purposes. A weather observer consists of a temperature and humidity sensors, a radiator consists of a general ON/OFF and a set point actuators, a presence Detector comprises of a presence sensor and a light controller comprises of a light control actuator.

Temperature and humidity sensors can report current temperature/humidity values, minimum/maximum measured values and the range used, while the light control actuator can control a light source, such as a LED by allowing the light to be turned ON and OFF and for example setting dimmer between 0-100% and counting the time the device has been on. A general actuator is very much similar. A presence sensor can report the current state of itself and count the number of "active" states detected. Set point actuator allows setting a specific target value. As an example, a temperature sensor and its resources are described in table 3.2.

### OpenWeatherMap

OpenWeatherMap is an open data source, which is utilized in the weather observer. Data is collected from thousands of remote sensors of over 40 000 weather stations all around the world. User can access the current weather data of over 200 000 cities. In the implementation, the data from Helsinki weather station is used. It is provided in JSON, XML or HTML format through APIs. An example of API respond in JSON format:

```
{
  "coord": {"lon":145.77,"lat":-16.92},
  "weather":[{"id":803,"main":"clouds","description":"brokenclouds","icon":"04n"}],
  "base":"cmc stations",
  "main": {"temp":293.25,"pressure":1019,"humidity":83,"temp_min":289.82,"temp_max":295.37},
  "wind":{"speed":5.1,"deg":150},
  "clouds":{"all":75},
  "rain":{"3h":3},
  "dt":1435658272,
  "sys": {
      "type":1,"id":8166,"message":0.0166,"country":"AU","sunrise":1435610796,"sunset":1435650870},
  "id":2172797,"name":"Cairns",
  "cod":200
}
```

Usage of the data source requires an API key, which is got by registering to the service. In addition, integrating the API to the implementation requires using a small abstraction layer, for example node-openweathermap can be utilized for that.

**Table 3.2: IPSO Temperature**

| Object Name | ID | Instances | Object URN |
|---|---|---|---|
| Temperature Sensor | 3303 | Multiple | urn:oma:lwm2m:ext:3303 |

| Resource | ID | Oper. | Mandatory | Type | Units | Description |
|---|---|---|---|---|---|---|
| Sensor Value | 5700 | R | Mandatory | Float | Defined by “Units” resource | Current measured sensor value |
| Min Measured Value | 5601 | R | Optional | Float | Defined by “Units” resource | The minimum value measured by the sensor since power ON |
| Max Measured Value | 5602 | R | Optional | Float | Defined by “Units” resource | The maximum value measured by the sensor since power ON |
| Min Range Value | 5603 | R | Optional | Float | Defined by “Units” resource | The minimum value that can be measured |
| Min Range Value | 5604 | R | Optional | Float | Defined by “Units” resource | The maximum value that can be measured |
| Sensor Units | 5701 | R | Optional | String | | Measurement units definition e.g. “Cel” for celsius |
| Reset Min and Max Measured Values | 5605 | E | Optional | String | | Reset the min and max measured values to current value |

### 3.2.2 Fuel Injection System

Fuel injection system is a complex system consisting of multiple parts, which ensure the fuel is used efficiently. In essence, fuel is pumped from the fuel tank to the engine under pressure and finally to the cylinders. In modern cars the fuel injection system is electronical: the operation is controlled entirely by a microprocessor, a miniatur computer called ECM. The system also uses indirect injection, where injectors spray fuel into a inlet valve rather than directly to combustion chamber to feed cylinders. This way the fuel mixes properly with air. [44]

Different operating conditions require different fuel/air mix and different amount of fuel. In order to control these factors, ECM monitors several sensors. It compares the sensor values to the predefined factory values and based on that determines the correct amount of fuel needed. The figure 3.1 illustrates the fuel injection system and its components. This use case models the following sensors and actuators:

- **Throttle Position Sensor (TPS)**: signals the throttle position, which is dependent on the position of a gas pedal. As the pedal is stepped on and off, the throttle is opened and closed accordingly letting correct amount of air flow to the engine. Computer uses this information to determine the required amount of fuel. [46]
- **Mass Airflow (MAF) sensor**: signals the air mass entering the engine. The information is needed to calculate the correct amount of fuel needed. [47]
- **Engine Coolant Temperature (ECT) sensor**: measures the internal temperature of the engine and signals the temperature changes to the ECM. The information is needed for the ECM to determine if the engine is cold, warm or overheating. [48]
- **Crankshaft speed sensor**: measures the rotational speed of the crankshaft. This information is also used to control fuel injection. Higher the engine speed, the more fuel is needed. [49]
- **Fuel pressure sensor**: measures the pressure of fuel in the rail. The pressure drop is signaled to the ECM, which restorates the pressure by controlling the pump valve. [44]
- **Fuel pump valve**: is an ON/OFF switch controlled by ECM. It is used to set the correct fuel pressure. [44]

- **Fuel injector**: is an On/OFF switch controlled by ECM. Injectors are opened and closed to feed the cylinders with a correct amount of fuel. [44]

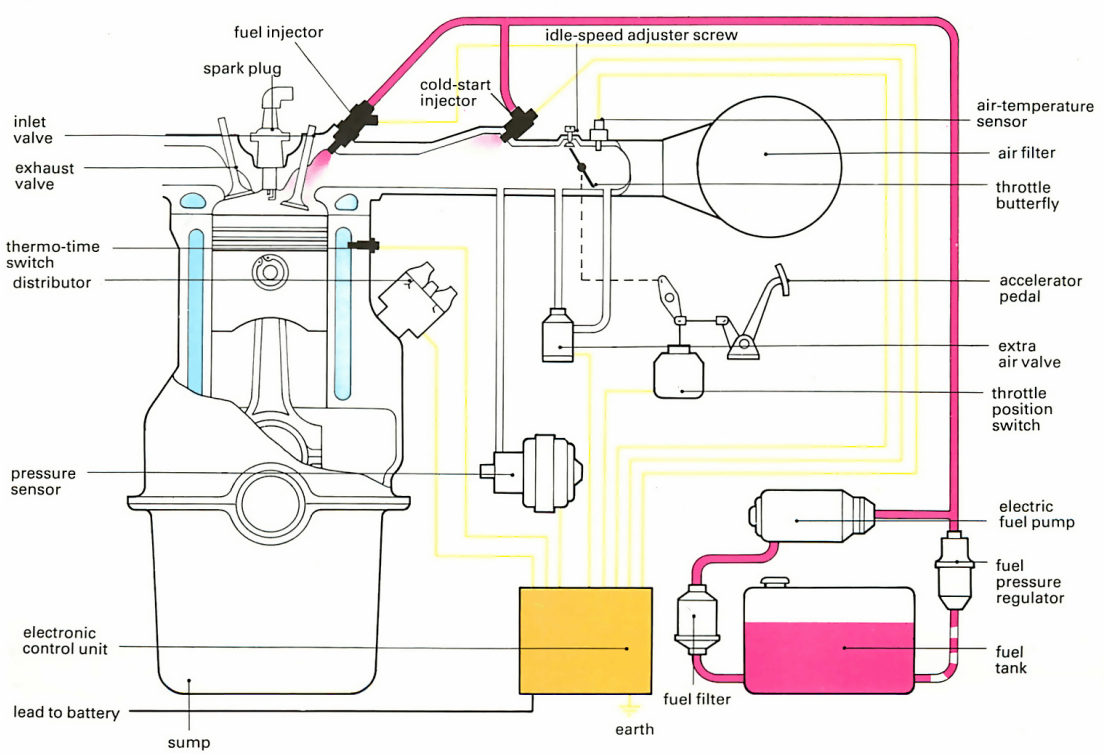



**Figure 3.1: Fuel injection system.** [44]

A complex device or system can be modeled with a IPSO composite object, which of resources are links to other objects. In a constraint environment a composite object may be a lighter solution than a large nested object in a sense that it enables observation of only linked object instances instead of the full object. Thus, it saves bandwidth. [34] Here, we chose IPSO 4000 ObjectInstanceHandler, which represent a common composite object. The table 3.3 shows the chosen linked IPSO objects and the corresponding real sensors. In reality, ECM contains both analog and digital inputs and outputs. Typically the sensors output a voltage value, which is proportional to the real measurement value. ECM digitize the data from analog sensors and calculates engine settings based on the voltage inputs. [50] The composite object with links is illustrated in figure 3.2.

**Table 3.3: IPSO objects & corresponding real sensors**

| IPSO Object | Real Sensor/Actuator |
|---|---|
| IPSO 3303 Temperature | ECT Sensor |
| IPSO 3316 Voltage | Throttle Position Sensor |
| IPSO 3316 Rate | Mass Airflow Sensor |
| IPSO 3346 Rate | Crankshaft Speed Sensor |
| IPSO 3323 Pressure | Fuel Pressure Sensor |
| IPSO 3306 Actuator | Fuel Injector |
| IPSO 3306 Actuator | Pump Valve |

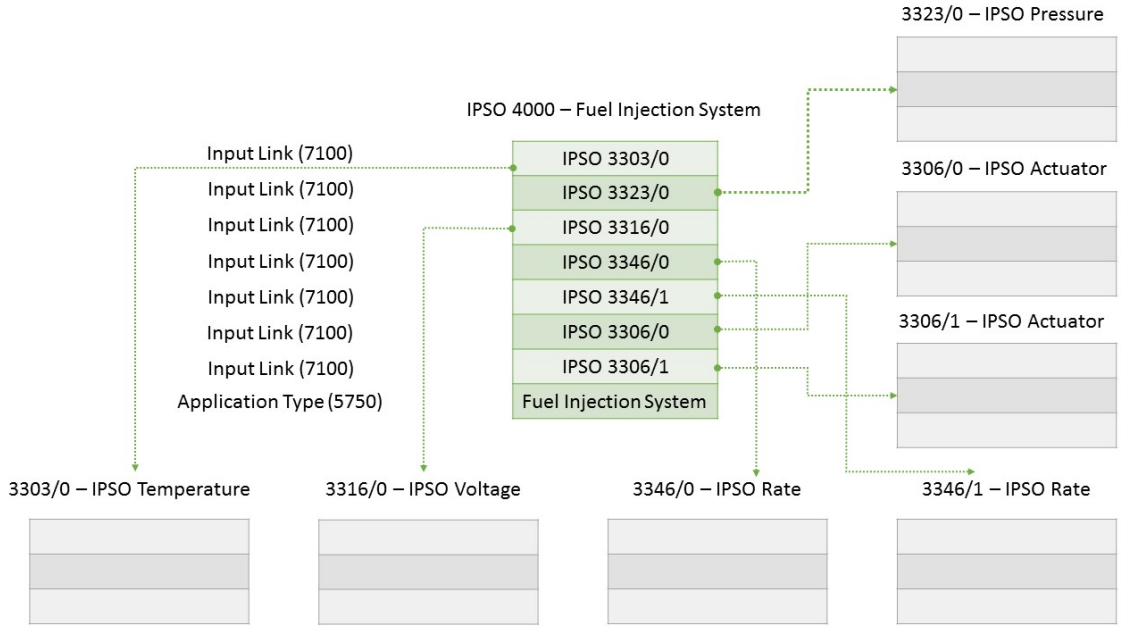



**Figure 3.2: The composite IPSO object with linked objects.**

### 3.2.3 Architecture

The overall architecture of the first use case consists of five components: LWM2M server and four LWM2M clients: weather observer, radiator, presence detector and light controller. The architecture is shown in figure 3.3. All devices are connected to and managed by LWM2M server, which maintains a registry of the available resources. In addition a weather observer and a radiator are communicating directly as well as a presence detector and a light controller.

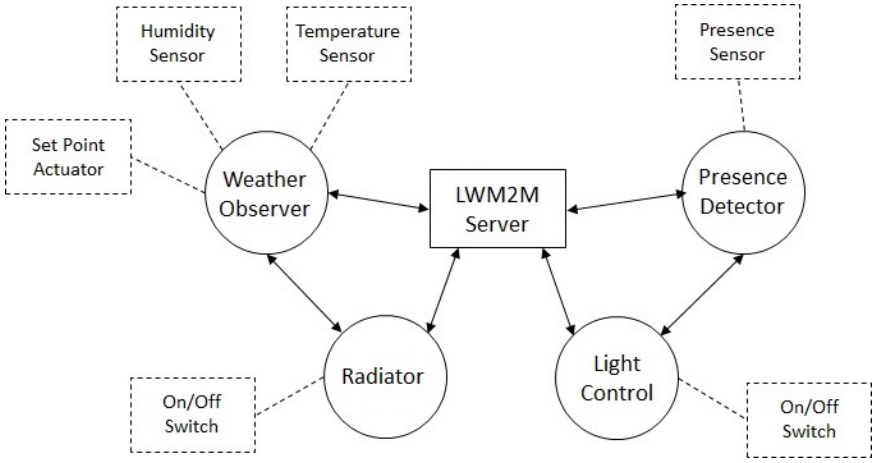



**Figure 3.3: Architecture of the first use case.**

The architecture of the second use case is illustrated in figure 3.4. The LWM2M server communicates with the fuel injection system operated by the ECM. ECM receives information from coolant temperature, throttle position, pressure, mass airflow and crankshaft speed sensors, and based on the current values sends commands to fuel pump valve to control the fuel pressure and fuel injector to control the amount of fuel injected to the cylinders.

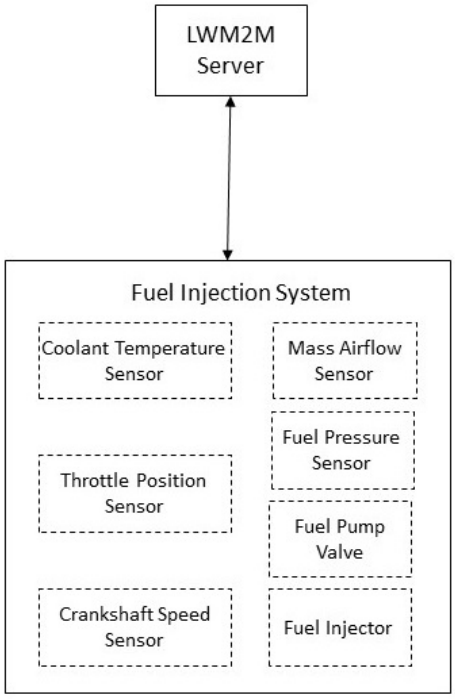



**Figure 3.4: Architecture of the second use case.**

# 4 Implementation

The implementation part of this thesis is about creating an emulator of virtual IoT devices in terms of two use cases: emulation of several simple devices and emulation of a single complex system.

The LWM2M client software applications created for virtual devices are implemented with node.js, which is a software platform for networking applications written in JavaScript. It is appropriate choice for the implementation, because of its asynchronous nature; it uses asynchronous events and non-blocking I/O making it very efficient. Except for network and file events, node.js applications are run single-threaded [52]. On the server side, Java based Leshan project is used. It provides a simple demo LWM2M server, which is utilized in this implementation.

The emulator is purposefully implemented in a virtual environment, where no physical devices are needed. The virtual testbed is created using Docker, for which appropriate configurations files need to be created. We also utilize docker-compose tool and Greenhouse tool presented in chapter 2 to be able to run multi-container applications and visualize the IoT scenarios.

This chapter presents first the software and hardware requirements for the implementation. Then, the actual implementation of the virtual IoT devices within the context of the use cases is described, including the object initialization, the communication logic and the virtualization.

## 4.1 Software & Hardware

This chapter presents the software and hardware required for the implementation. In terms of software, the virtual devices are implemented by writing applications in JavaScript for nodejs platform. The applications define the objects representing the virtual sensors and actuators and all the functionalities of the devices. One of the main software library used in the implementation is JavaScript written coap-node, which provides all the basic functionalities required for the LWM2M clients. The LWM2M server used is a Java written Leshan demo server, which supports basic server side functionalities and is easy to set up, serving well the emulation purposes.

The hardware requirements for the implementation are minimal.

### 4.1.1 Coap-Node

Coap-node [53], is a simple open-source node.js implementation of LWM2M for M2M/IoT client devices under MIT license. The module supports CoAP protocol along with the IPSO model of smart objects consisting of several APIs, which handle the client/server interaction through LWM2M interfaces. The implementation is primarily designed to work with node-shepherd [55] based server. However, in this thesis work the Leshan based server was chosen to manage LWM2M clients, because of its more advanced capabilities. This has required making some adjustments to the client side in order to be compatible with the Leshan sever.

The core of the module relies on node-coap [54], another open-source project, which is a CoAP client/server library written for node.js. It implements the fundamentals of CoAP -defined communication, including the CoAP message header, message types, options, request/response codes and retransmission methods. Coap-node also utilizes node.js smartobject module [56] for organizing resources on device.

While the work is still in progress, the library already implements several basic functionalities related to LWM2M client Registration, Information Reporting and Device Management and Service Enablement Interfaces, such as registration/deregistration procedure, support for basic read/write/execute methods, resource initialization, resource discovery/lookup interface, observation and write attributes method [53]. Nevertheless it is still missing the bootstrap interface and the following capabilities:

- TLV encoding/decoding
- Support for Create/Delete methods
- DTLS security over UDP packets

The missing TLV encoding/decoding is not a critical problem for the implementation, since JSON format is supported and can be used to encode/decode data objects and multi-instance resources instead. Create and Delete methods are relevant functionalities of LWM2M, but don't have high priority in the implementation. There is currently no proper DTLS implementations for node.js, which is why security features are ignored in this implementation and "no security" -mode is used instead. In reality, the security issues are an important part of real IoT implementations, but

here the focus is mostly on the device logic.

### Resource planning

The simplistic way of resource planning is one of the key benefits of the coap-node module. Since it follows the LWM2M and IPSO specifications, it is easy to initialize the basic objects, such as a standard Device Object or IPSO temperature sensor, and their resources by using the hierarchical object structure and predefined IDs. Depending on the target resource, one needs to determine the allowed operations (read/write/execute) in order to be able to handle the LWM2M server requests and responses correctly. For example, initializing resources 'Sensor Value' and 'Unit' of a temperature sensor may look like this:

```
cnode.initResrc(3303, 0,
   5700: {
      read: function (cb) {
         var tempVal = gpio.read('gpio0');
         cb(null, tempVal);
      }
   },
   5701:'Cel'
});
```

In this case the sensor value (ID 5700) is initialized with the read method and the unit (ID 5701) is initialized as a primitive value. [53]

### Adjustments & Improvements

The library has been slightly modified during this thesis work to support the Leshan server instead of the "default" coap-sheperd server. Also in the beginning of this thesis work the library didn't support Create operation, which has been implemented. The edits to the source files coap-node.js/reqHandler.js are the following:

- Function *CoapNode.prototype._updateNetInfo()* is used to find network information of the client, such as IP address, MAC address and port number. However, inside a Docker container the network module can't find any active networks and the registration fails. Thus, a new function *network.get_interfaces_list(callback)* is added to find the network information inside a container.

- Function *CoapNode.prototype.register(ip, port, callback)* is used to register the LWM2M Client to the Server. Before the registration request is sent to the Server, location path should be modified to self.locationPath = '/rd/' + rsp.headers['Location-Path'], where rd refers to the resource directory. Without rd in the path, the Server can't find the Client and registration fails. In addition, function *startMultiListener(self, callback)* needs to be called inside register function (see also number 5).

- The library supports JSON-format, but before it can be used and interpreted by the Server correctly, it needs to be registered with the function *coap.registerFormat('application/json', 1543)*, which has been added to the library.

- To support Create operation, function *CoapNode.prototype.createInst(oid, iid, value, callback)* was implemented. It takes the target object ID, instance ID and resources wrapped in an object as parameters and creates a new object instance with defined resources. Also a corresponding request handler was created with a function *serverCreateHandler(cn, req, rsp)*, where cn is the CoapNode object, req is the request from the LWM2M server and rsp is the response, which is sent to the server in the end. The request handler first decodes the JSON formatted payload and then initialize the new object instance with given resources.

- In the implementation, multicast functionality is very important. Hence, a function *CoapNode.prototype.multicast(path, method, value, callback)* was created. It takes the object path (eg. /3303/0/5700), method (eg. PUT) and value (payload) as parameters and creates a multicast request. Also a function *startMultiListener(cn, callback)* was created to make the LWM2M clients to listen the multicast address.

### 4.1.2 Leshan

Leshan is a java based Lightweight M2M implementation under Eclipse Public License, which provides libraries for LWM2M client and server. The development started in 2013 by Sierra Wireless, but continued as Eclipse project in 2014. Leshan relies on Californium project for CoAP and Scandium project for DTLS implementation, which cover the basic functionalities of CoAP and security with Pre-Shared-Key, Raw-Public-Key and X.509 options. In addition, it supports LWM2M and IPSO objects. [59]

Leshan project currently includes features, such as client initiated bootstrap, registration/deregistration, capability to read, write, execute, create, delete and observe objects and TLV/JSON encoding/decoding. The server also implements Client Registry, which stores all the registered clients, and Security Registry, which stores the required security information. The registered clients are assigned a unique client ID, which is used to identify the clients. [58]

### Server Demonstration

The project also includes a ready-to-use server demo, which is utilized in this thesis. The demo server can handle read/write/execute operations along with the observation requests in several data formats. It contains a simple web UI, which manages user commands, visualizes the communication between the client and server and shows CoAP messages. [58] The web UI is illustrated in figure 4.1.

The demo server doesn't support resource discovery/lookup interface, which sets some limitations to the implementation. With resource discovery, devices could find all the public resources in a quite easy way, which would be useful for the device-to-device communication. The lookup interface enables making queries about spesific resources or endpoints. This limitation is overcome by using CoAP multicast, but it is a waste of bandwidth.

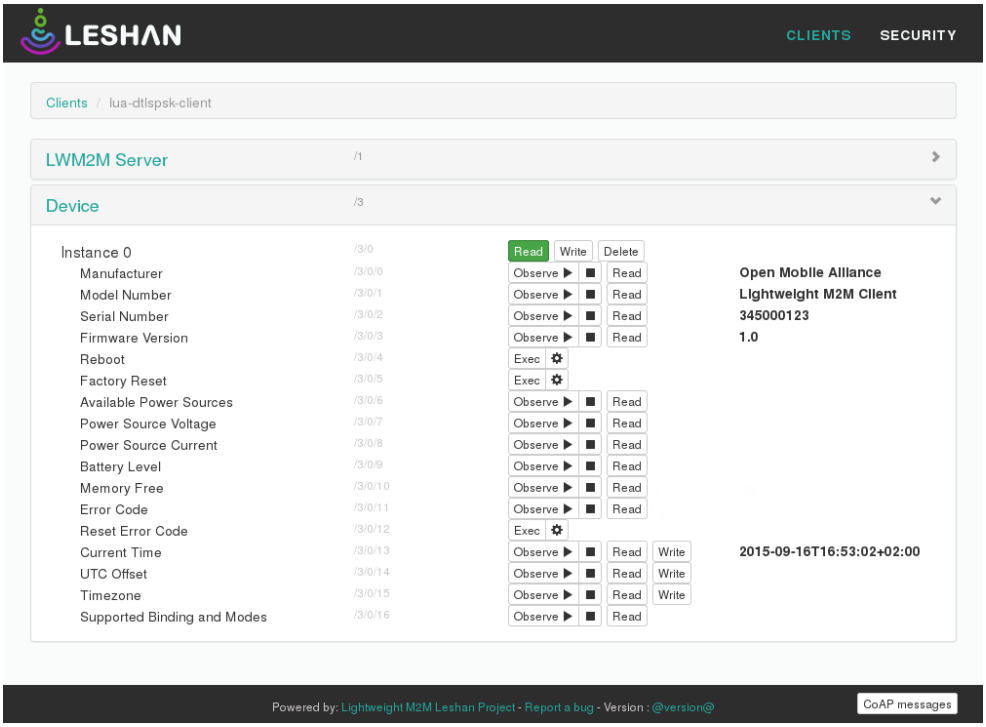



**Figure 4.1: User Interface of Leshan demo server.**

### 4.1.3 Hardware

The hardware requirements of the implementation are minimal. Basically any computer with sufficient system resources (memory, disk space, processing power) should be enough for the implementation. Also Internet-connection is required. We use Dell-laptop with 64-bit Ubuntu 14.04 LTS operating system and Intel i7-series processor.

## 4.2 The Implementation of IoT Devices

The implementation of the virtual devices is divided into three categories: object initialization, communication logic and virtualization. The first part is about implementing the IPSO and LWM2M objects chosen to represent the sensors and actuators and manage them. The data read from the sensors or wrote to the actuators is accessed and organized through these objects, which is why resource planning is very important. Nevertheless, the most important aspect of the implementation is the creation of device logic. While the devices are virtual, they are supposed to act as real ones with the desirable functionalities. The communication logic includes for example realizing the actions taken on the executable resources and planning the operational chains between the devices, thats is, how the actions of one device effects other devices and trigger another action. The third part is about virtualizing all the components of the emulator by dockerizing them (enable use of Docker) and using for example Greenhouse tools. The next sections describe the object initialization, the communication logic and the virtualization of the two use cases presented in chapter 3 in more detail.

### 4.2.1 Object Initialization

All devices are set up by initializing first the needed LWM2M and IPSO objects and resources. The LWM2M objects are the same regardless of the device type, while the IPSO objects and their resources are chosen based on the specific application type. All resources can be defined as readable, writable or executable accordance with the LWM2M and IPSO specifications. For example, sensor values are readable, while the status of actuators is readable and writable. Executable resources trigger an action, which usually involves separate function calls.

From the set of default LWM2M objects Security, Server, Device, Location and

Connection monitoring objects were chosen to demonstrate the basic management features IoT devices may have. From the security object only the resource 0, Server URI, is initialized, since the current implementation doesn't support any security mechanisms. Server object enables for example defining the lifetime value of a device and triggering a device update, whereas Device object provides means to reset and reboot the device. Location and Connection Monitoring objects provide information on current device location and networking, such as IP addresses of the device and gateway.

The basic IPSO objects can be easily initialized, but the composite object used for the fuel control system is more difficult. It introduces a new data type, "ObjLnk", which is an object link used to refer to an instance of a given object [32]. Since object link is relatively new data type, the LWM2M libraries used in the implementation don't currently support that. Also, the IPSO Alliance haven't yet validated any official composite object, nor resources for the input links. Thus, we register a new composite object with ID 4000 and resources with IDs 7100-7106 to the LWM2M server. The input link resources support "String" as a data type instead of "Objlnk" and contain a link in the form of objectID/instanceID pointing to the sensors and actuators relevant for the use case. Now, when the user reads a resource of a composite object from the LWM2M server, the device will respond back with the instance of the linked object by redirecting the GET message to it.

### 4.2.2 The Communication Logic

The communication methods, functionalities and the operation process of the devices are referred here as communication logic. The communication logic of the devices varies a bit depending on the use case and for that reason the different use cases are addressed here separately.

In general, the LWM2M functionalities of the devices are always the same regardless of the use case. As the LWM2M client software applications start running, the virtual devices first register to the LWM2M server with predefined IP address and CoAP default port, after which they are ready to communicate with the server and other devices available on the same network. The termination of the process, that is, the received SIGINT or SIGTERM signals, will cause the device to issue a deregister operation. As SIGINT is generated by CTRL+C user input, SIGTERM does the same inside a docker container.

### Use Case 1: simple IoT devices

The first use case is about an emulation of simple virtual IoT devices presented in chapter 3. All the devices are connected to the LWM2M server and managed by it. For the devices to be able to communicate to each other, they need to first discover each other's available resources. The CoAP specification provides means for this with resource discovery and lookup operations; device can send a GET message to the LWM2M server with a URI of well-known/core and receive links to all available resources or device can query spesific information of another device. However, Leshan server doesn't fully support these functionalities, which is why CoAP multicast is used instead.

After registering to the LWM2M server, devices are ready to communicate with the it. A weather observer has two modes: random data mode and real weather data mode. If the previous one is used, the temperature data is randomly generated between values 0 and 30. If the latter mode is used, an openweathermap data source, presented in chapter 3, is used. Whenever the current temperature value is read by the LWM2M server, a weather observer compares it to the set target value and if necessary, sends a PUT /3306/0/5700 message with a value '1' or '0' to CoAP multicast address 224.0.1.187. The set target value is 15 celsius by default. If the temperature value is above it, weather observer demands the radiator to be turned OFF (value '0') and in the opposite situation to be turned ON (value '1'). In the case, where temperature hasn't changed much, no multicast message is sent. A radiator device receiving this message changes it status to ON or OFF based on the control message of weather observer.

The communication logic between the presence detector and a light controller is similar. The status of a presence detector randomly varies between 'true' and 'false' to simulate the scenarios, where the room is occupied with people or is empty. Whenever the status is read by the LWM2M server, the detector will command the light controller to turn the lights ON or OFF by multicasting a PUT /3311/0/5850 message with a value '1' or '0' respectively. For example, if the status of the presence detector has changed from false to true, detector will send value '1' to the light controller, requesting the lights to be turned ON.

### Use case 2: fuel injection system

The second use case is an emulation of a complex engine fuel injection system, which consists of several sensors and actuators described in chapter 3. The system is

connected to the LWM2M server and managed by it. Here, the focus in on the communication logic inside a complex system.

In a real fuel injection system, ECM receives a lot of information from several sensors in real time and based on the input values, determines the amount of fuel needed. It then sends commands to different actuators that control the fuel injection. Typically, the sensors convert the measurement units, such as celsius, into volts for the ECM to understand the input signals. ECM has reference values programmed in the factory and it compares them to the input signals to determine the fuel need in different scenarios. The reference values for different scenarios are gathered in table 4.1 [50] [47] [48] [49] [51].

**Table 4.1: Reference values for ECM**

| **Sensor** | **Quantity** | **Reference values** |
|---|---|---|
| Coolant Temperature Sensor | Temperature | Cold: 0 – 10 C<br>Warm: 10 – 70 C<br>Hot: 70 – 100 C |
| Throttle Position Sensor | Voltage | Close: 0,5 – 2,0 V<br>Semi-open: 2,0 – 3,0 V<br>Open: 3,0 – 5,0 V |
| Mass Airflow Sensor | GPS (grams per second) | Idle: 0 – 15 g/s<br>Throttle: 15 – 100g/s<br>Full-throttle: 100 – 160g/s |
| Crankshaft Speed Sensor | Rotational speed | Idle: 0 – 600 rpm<br>Throttle: 600 – 4000 rpm<br>Full-throttle: 4000 – 5000 rpm |
| Fuel Pressure Sensor | Pressure | 30 – 80 psi |

For simplicity the reference values are rounded. Also the logic of how the different values are effecting the fuel amount is modeled in a very simple way. The idea is to just show that the measurement values increase/decrease about in proportion and these changes together has an effect on the fuel amount. For each sensor, there are three categories of values (see table 4.1), each of which adds a certain amount of time for injectors to stay open. Based on this assumption, we create the following rules:

- ECT: +3s (Cold), +2s (Warm), +1s (Hot)
- TPS: +3s (Open), +2s (Semi-open), +1s (Closed)
- MAS: +3s (Full-throttle), +2s (Throttle), -2s (Idle)

- CSS: +2s (Full-throttle), +2s (Throttle), +1s (Idle)

Thus, for example starting a car with cold engine and accelerating with open throttle adds 3s to the injection time, while the idle speed of crankshaft adds only 1s to the injection time. The airmass has a bit different effect; a rich air-fuel mixture (too much air) adds 2s to the injection time, while a lean mixture (too little air) decreases 2s the injection time. Mixture in between adds 1s injection time. In reality the logic is much more complex and for example the air-fuel mixture is maintained by controlling the throttle opening not only manually (driver presses gas pedal), but via throttle controller.

Since the focus of this use case is not on the real operation of a fuel injection system, but on modeling a complex system with IPSO objects in context of IoT, the logic can be very simple and plain. We assume a somewhat linear relation between the different sensor values; as throttle opens, also the engine speed, coolant temperature and airmass increase and the other way around. To simulate the data, we use a linear function:

y=kx/10, where k is randomly generated slope [1,5], x is time in seconds.

The resulted y is a percentage value, which is multiplied by the maximum sensor value to get the "current" sensor value. When 100% is reached the time t is reseted. After reading all the values from different sensors, ECM determines the amount of fuel needed by calculating the fuel injection time and sends a control message to the fuel injector to open and finally close after the time calculated expires. The fuel pressure sensor and pressure control valve are needed to constantly maintain the correct pressure in the fuel track. If the pressure drops, ECM sends message to the control valve to close it. In this implementation, the amount of pressure is randomly generated between values [20,90], where the normal pressure is between 30-80 psi.

The values of sensors and actuators can be read from the LWM2M server by issuing a read request to the composite object's resources. When these resources are read or wrote, the request is redirected to the actual sensor object, eg. temperature object 3303/0 or voltage object 3316/0, and the response contains the sensor data behind the link.

### 4.2.3 Virtualization

Since the sensors and actuators are not real devices, we can utilize the benefits of virtualization, which enable us to scale the amount of devices and use the emulator

regardless of the host machine's operating system and cumbersome, physical testbeds. One can simply use any computer with installed Docker. Greenhouse tool provides some special features, which visualize the implementation a bit better, but it is not an essential part of the emulator.

### Testbed for Virtual Devices

We create the testbed for virtual devices by using Docker. This requires understanding the networking of Docker more deeply. Docker automatically creates a docker0 network interface on the host system and a subnet 172.17.0.0/16 with the gateway address of 172.17.0.1. For each running container, it allocated an IP address from its address pool and a MAC address. An example below shows the "bridge" network, which represents the docker0 network and information of two running containers.


```
{[
    {
        "Name": "bridge",
        "Id": "f7ab26d71dbd6f557852c7156ae0574bbf62c42f539b50c8ebde0f728a253b6f",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Config": [
                {
                    "Subnet": "172.17.0.1/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
        "Containers": {
            "3386a527aa08b37ea9232cbcace2d2458d49f44bb05a6b775fba7ddd40d8f92c": {
                "EndpointID": "647c12443e91faf0fd508b6edfe59c30b642abb60dfab890b4bdccee38750bc1",
                "MacAddress": "02:42:ac:11:00:02",
                "IPv4Address": "172.17.0.2/16",
                "IPv6Address": ""
            }
        }
    }
]
```


If all the components are run inside docker containers and there is no need for communicating with the external networks, the networking is very straightforward. If though, containers need to be reachable from the outside world, appropriate ports must be exposed and mapped correctly. For example the LWM2M and Web servers running inside a container need to expose UDP port 5683 and TCP port, such as 8080, if clients running outside containers need to interact with the LWM2M and

Web servers. The communication goes through docker0 interface. It is also possible to run the clients inside containers and the server on localhost. In this case, the server needs to be started at the docker gateway address, to which the clients connect. The figure 4.2 illustrates the networking between containers and the outside world.

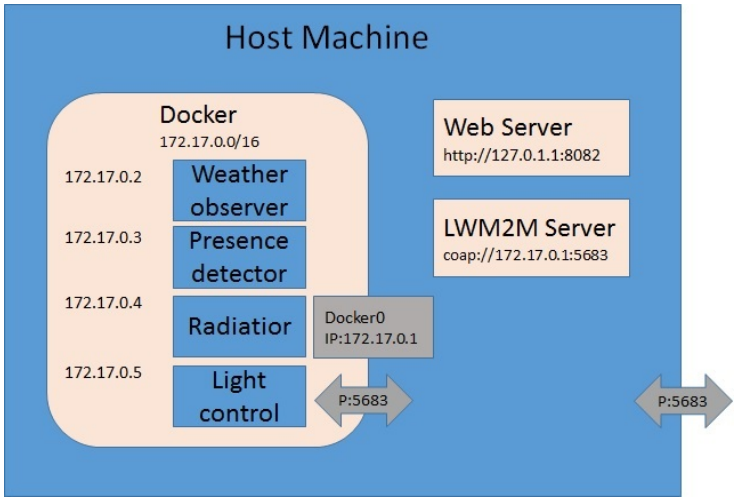



**Figure 4.2: Networking of Docker.**

Dockerizing the devices involves creating appropriate Dockerfiles for configuration and building the Docker images based on them. The application also needs a package.json file, which defines all the required library dependencies and the start script. Each device has its own Dockerfile, which contains a set of instructions for building the docker environment. In this implementation we build the Docker images of virtual devices upon the base image called 'node:4.6'. It is one of the official Docker images, which can be used to build the node.js platform. Dockerfile also includes instructions to install the dependencies according to package.json file and expose ports if necessary. For virtualizing the Leshan demo server, we use an official leshan-docker image, which builds minimal runtime environment for the java server to run properly. After images are created, the containers can be run. As default, the process inside a container is started in foreground mode meaning that the console is attached to the standard input, output and error of the process.

Furthermore, we want to use Docker-compose, which is a tool for running multi-container Docker applications. In other words, we can run all the different devices and server in their own containers simultaneously within one tool with one command. Using docker-compose requires creating a yml-file, which defines the different "services" with the information about how they are built, which ports are exposed and what are the dependencies between the services. With docker-compose the amount

of devices can easily be scaled with the scale command. We use this functionality to test the scalability of the application.

### Visualization of the implementation

The implementation can be visualized using Greenhouse tool. Particularly, we utilize a Jekyll and Tesla add-ons described in chapter 2. The setup requires creating an appspec.json and task.json files. The former one defines the components we are using including component IDs, correct task files and the amount of instances we want to create. The latter one defines for example the used Docker image with Greenhouse configurations, tools and polarity of the component, which determines the layout of the connection links between the components. Each device has its own task file.

# 5 Evaluation

This chapter presents the test environment and evaluates the implemented emulator in terms of functionalities and performance. Evaluation of functionalities includes discussion on, how well the design requirements presented in chapter 3 were fulfilled and what are the remaining issues. Evaluation of performance is about testing the scalability of the implementation and presenting the findings. In addition, the IoT traffic is analyzed regarding of bandwidth usage and size of data packets.

## 5.1 Test Environment

The use cases are executed in a simple test environment, which consists of one host machine and a WiFi connection. The host machine used is a laptop with 64-bit Ubuntu 14.04 LTS operating system, Intel® Core™ i7-4800MQ processor (2.70 GHz x 8) and 15,6 GiB memory. Each virtual component of the emulator is run inside a separate docker container (virtual machine) in the same network. We use the local WiFi connection with speed of 1000 Mb/s.

The use cases are executed by starting first the LWM2M server inside a container or on localhost, and then running the created docker images for virtual devices relevant for each use case. After the components are up and running, user can play with the emulator by reading, writing or executing resources through a Leshan web user interface on browser.

## 5.2 Evaluation of Functionalities

In this section, the functionalities of the implementation are evaluated in terms of the design requirements described in chapter 3. The focus is on the execution of the two use cases and how the design requirements are fulfilled in those in the test environment. Also the remaining issues of the implementation are briefly covered.

### 5.2.1 Fulfillment of Functional Requirements

The fulfillment of functional requirements is evaluated in table 5.1. Each functionality is then descirbed in more detail in terms of, how they are implemented or why the functionality is still missing.

**Table 5.1: Fulfillment of Functional Requirements**

| #number | Functionality | Fulfillment |
|---|---|---|
| 1 | Registration/Deregistration | Yes |
| 2 | Resource initialization | Yes |
| 3 | Read/Write/Execute | Yes |
| 4 | Create/Delete | Yes/No |
| 5 | Observe/Notify | Yes |
| 6 | JSON/TLV format | Yes/No |
| 7 | Device-to-Device communicatione | Yes |

1. The implemented devices are able to register and deregister according to the LWM2M specification. As the device applications are started, they send POST message with the required query parameters, such as endpoint name and default lifetime value to the LWM2M server. Respectively, they send a DELETE message to the LWM2M server, when the applications are terminated.

2. Resource initialization is implemented as it is defined in the coap-node library. Both LWM2M objects and IPSO objects are supported. The implementation is only lacking of initialization of resource instances, which is a less important feature.

3. Reading, writing and executing resources of both single-instance and multi-instance objects are implemented. Also reading the whole object instances is supported. Multi-instance support requires registering the correct data format, such as JSON.

4. Create functionality is partly implemented. It works only with JSON data format. Depending on the Leshan version, TLV format may be required and in that case, it won't work. Delete functionality is not implemented because of the limited time.

5. Devices are able to observe resources and object instances and send notifications to the LWM2M server. The implementation requires that notifications are sent

as confirmable messages, which may not be optimal solution in all cases, since it creates more traffic.

6. JSON format is supported by adding minor modifications to the coap-node library and registering the format correctly as mentioned before.

7. Devices are able to communicate with each other by sending multicast messages. Multicast is appropriate solution, when reliability is not required. Multicast messages are noncorfimable messages, which sets some limits to what can be multicasted.

### 5.2.2 Remaining Issues

The emulator still has some issues and unsupported functionalities, which could be later added to the implementation. Here, we list some features, which still require development and would improve the operation of the emulator:

- TLV data format
- Objlnk data type
- Create/Delete methods
- Resource Discovery/Lookup Interface
- Observation between devices
- DTLS support

TLV data format is very useful for IoT communications, since it is a binary format reducing the size of the transmitted data. Objlnk data type is needed for the composite objects. As TLV format is supported, also the implemented Create method should work fine. Delete method would be useful to remove resources and objects.

The emulator would benefit mostly from the resorce discovery and lookup functionalities. Devices would discover each other's available resources easier than using multicast. In the current implementation, devices don't directly observe each other's resources, but instead they send data (via multicast), when observation is started from the LWM2M server.

Obviously, security is an important aspect in real life IoT scenarios. Thus, implementing the DTLS support would be mandatory at some point.

## 5.3 Evaluation of Performance

In this section, the performance of the emulator is evaluated in terms scalability and network parameters, such as bandwidth usage. The main focus is on the scalability tests and the results of optimization done based on the issues found during the testing.

### 5.3.1 Scalability

The scalability tests are made by using docker-compose with scale option. The test scenario builds around one weather observer and presence detector, and several radiators and light controllers connecting to the LWM2M server. We chose to have only one weather observer and presence detector, since they are the ones sending multicast messages and thus, stressing the network mostly. Real life situations could involve a building with many radiators, which get information from one "central" weather observer and a big hall with multiple lights that are controlled based on the information provided by presence detector.

The test is executed by trying first to connect 50 devices, from which one is a presence detector, one is a weather observer and half of the rest devices are radiators and half light controllers. With this kind of logic, the number of devices is increased to 100, 150, 200 and so on. We also use two different loads; load0 is created by having only one device of each type sending periodically (observing a resource) data to the network and load1 is created by having at least half of all the devices sending periodically data to the network. The results of the test are collected to the table 5.2.

As we can see from the table 5.2, the maximum amount of devices the system can maintain in the network with load0 was 600. With a larger number, the docker-compose crashed or the devices dropped from the network unexpectedly. When the test was repeated with the load1, the maximum amount of devices was decreased to 450. Few devices were dropped from the network, but otherwise the emulator was working as expected.

A minor issue noticed already during the test was that the amount of devices actually connected to the LWM2M server didn't always match the number of launched devices. For example, when 200 devices were launched, only 197 devices were connected. This problem relates to the naming procedure of the devices. We use a method, where the device name consists of a string, such as "radiator", and an integer number between

**Table 5.2: Scalability Results**

| Number of Connected Clients | Successful Registration load0 | & Communication load1 |
|---|---|---|
| 50 | OK | OK |
| 100 | OK | OK |
| 150 | OK | OK |
| 200 | OK | OK |
| 250 | OK | OK |
| 300 | OK | OK |
| 350 | OK | OK |
| 400 | OK | OK (1 exit) |
| 450 | OK | OK (4 exit) |
| 500 | OK | X |
| 550 | OK | X |
| 600 | OK | X |
| 650 | X | X |
| 700 | X | X |

**Load0 = One device of each type communicating periodically (observing a resource)**
**Load1 = At least half of the devices connected communicating periodically.**

1 and 10000. However, the LWM2M demands that the endpoint name is unique, which may not hold in our case. If multiple devices try to use the same endpoint name, the latest one overwrites the previous one. Increasing the maximum integer number to 50000 improved the situation, but names are still not absolutely unique.

### Optimization

The maximum number of devices connected successfully was expected to be much larger. For that reason, the implementation of emulator was further investigated and optimized to improve the scalability. Some issues discovered related to the docker image size, the memory usage of containers and system load.

The easiest means to optimize the emulator was to change the base image used in the Dockerfiles of devices. As the devices require the nodejs platform, we still use the node base image, but a different version called node:slim. It implements the minimal requirements needed to run nodejs applications. This way, the docker image size was reduced from about 670 MB to 230 MB. The memory usage of the containers was

even more notable factor in the performance of emulator. The default memory limit of the containers is 15.58 GiB and the containers of the devices were actually using about 15-19 MiB. The emulator was optimized by limiting the memory usage of containers to 4 MB (the minimum amount of memory to allocate in Docker), which seemed to be suitable for devices.

As the scalability test was repeated, we chose to use a tool called dstat at the background to measure a load average of the system. Load average is a useful performance metric, which indicates the overall amount of computational work performed by the system. It includes all the processes and threads waiting for resources (I/O, networking etc.). Tool dstat shows three values for load average, which refer to the system operation over past one, five and fifteen minutes. We focus on the 1-minute load average. As we have an 8-core CPU the upper-bound for the system load average is 8. If the number exceeds it, the system is overloaded. This may result in unstable and unpredictable behavior of the system. The figure 5.1 shows the 1-minute load average measured over 1000 s interval with 600 devices trying to register to the LWM2M server simultaneously and being idle afterwards.

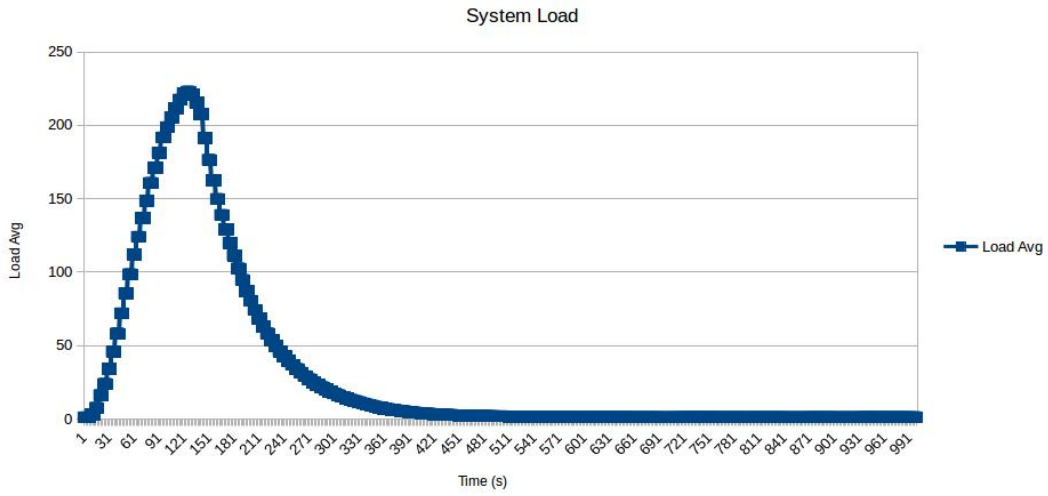



**Figure 5.1: Graph of load average of the system.**

The figure 5.1 shows that the load average rises extremely high (peak over 200), when the devices start to register to the Server. As we try to connect hundreds of devices at the same time, the system load doesn't spread over time much, but peaks sharply instead. This was the main problem in the scalability tests and for this reason a script was written to launch 100 devices at the time and then wait 15 seconds before launching another 100 devices until the desired number is reached.

Another problem stemmed from the multicast. As stated before, multicast wastes the bandwidth and stresses the network unnecessarily, which was also confirmed by the scalability tests. Having over 600 devices registered, from which two of them sending multicast messages regularly, caused the load average jump high, whenever multicast was triggered. This also explains, why devices sometimes dropped from the network at a random time; the unstable behavior of the system resulted in errors. The unicast messages didn't have notable effect on the system load. For this reason, only the maximum number of registered clients without any load was measured in the second scalability test. The results of second scalability test after optimization are in table 5.3.

**Table 5.3: New Scalability Results**

| Number of Connected Clients | Successful Registration |
|---|---|
| 700 | OK |
| 800 | OK |
| 900 | OK |
| 1000 | OK |
| 1100 | X |

The table 5.3 shows that we managed to launch the maximum of 1000 clients without any errors. Nevertheless, the actual amount of connected clients was 980 in that case partly due to naming procedure explained before and parly due to the still large system load. If we would like to have better results, we would need to increse the sleep time in the script and eventually, more capable hardware.

### 5.3.2 Traffic Analysis

In addition to evaluating scalability, we analyze the traffic by measuring network parameters, such as bandwidth usage of devices, and investigating further the data packets. To measure the traffic, we run LWM2M server on localhost and devices inside docker containers. This way, we can monitor the incoming and outgoing traffic of docker0 interface with the gateway address of 172.17.0.1. The information of bandwidth usage and packets can be used to analyze the performance of the emulator.

#### Bandwidth Usage

We use applications, such as iftop [60] and wireshark [61] to monitor the traffic in

real time. The former one measures current bandwidth usage per interface and per connection (pair of hosts). The latter one provides a lot of information for example about the packets and protocols, and also useful statistics from the traffic, such as graphs of current traffic load.

With iftop we measured the peak data on docker0 interface over 40 seconds time interval in four scenarios: 4, 10, 15 and 20 devices sending and receiving data periodically. The figure 5.2 is an example of iftop interface showing the scenario with 10 devices. The peak values of bandwidth usage at the bottom of the figure are measurements over a 40 seconds interval, while the data rates at which traffic is sent and received are measurements over 2, 10 and 40 seconds. The results from iftop are gathered in table 5.4. It shows the peak traffic separately for sending and receiving data (Tx, Rx) and the total values.

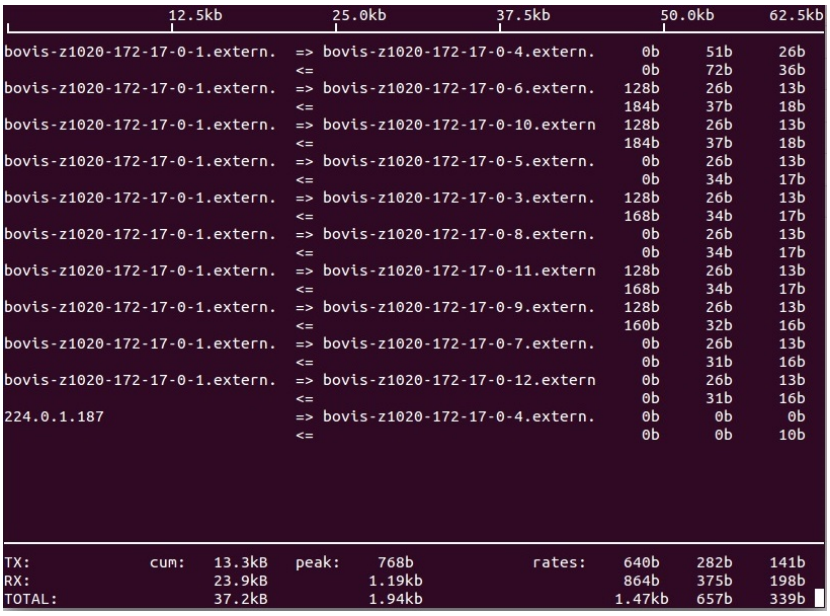



**Figure 5.2: The current bandwidth usage of docker0 from iftop.**

**Table 5.4: Peak Bandwidth usage of docker0 interface.**

| Number of Devices | Tx (peak) | Rx (peak) | Total (peak) |
|---|---|---|---|
| 4 | 384 bits | 744 bits | 1.128 kbits |
| 10 | 768 bits | 1.23 kbits | 1.998 kbits |
| 15 | 1.12 kbits | 1.73 kbits | 2.85 kbits |
| 20 | 1.38 kbits | 2.05 kbits | 3.43 kbits |

From the table 5.4 we can see that the peak bandwidth usage increases quite evenly, when the number of devices grows. Even though we had only maximum of 20 devices in the test generating quite small amounts of traffic, we can say that increasing the number of devices effects on the performance of network. As the IoT devices hog the bandwidth, there is not so much left for other services and devices. In the scalability test with hundreds of devices, the usage of the emulator seemed to slow down a bit. Obviously the influence is bigger in a network, which of bandwidth is more limited.

From WireShark we got a graph of the current traffic load in bits/s as shown is figure 5.3. It illustrates the traffic pattern; instead of continuous data flow with bulk data, IoT devices sent several small packets at certain times, while otherwise were idle. This may cause somewhat unpredictable bandwidth usage. This is a typical characteristics of IoT traffic, which sets some new requirements for the future networks. Especially, when there are hundreds of devices transferring data, the peak values may increase relatively high at times.

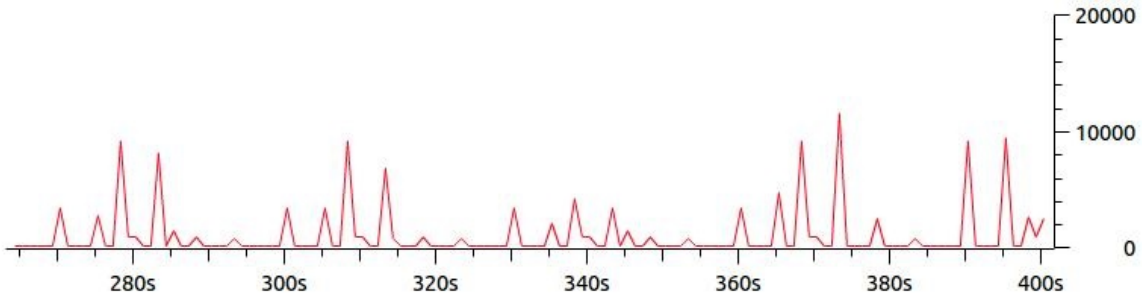



**Figure 5.3: Traffic load from WireShark with 10 devices.**

### Packets and Protocol stack

By investigating the traffic more closely, we can confirm that IoT devices indeed generate numerous amount of small packets instead of bulk data and follow the IoT protocol stack. From the figure 5.4 we can see that the length of all packets transmitted are in the range of 40-79 bytes.

According to CoAP specification [19] CoAP packet should fit in a single IP packet to avoid fragmentation. For IPv4 the minimum value for maximum transmission unit (MTU) is 576 bytes, whereas IPv6 has MTU of 1280 bytes. If there is no information about the size of headers, good upper bounds are 1152 bytes for message size and 1024 bytes for payload size. The packet size could be even less to avoid the possible fragmentation of lower layers, which may have their own MTU limits.

| Topic / Item | Count | Rate (ms) | Percent |
|---|---|---|---|
| ▼ Packet Lengths | 446 | 0.001115 | |
| 0-19 | 0 | 0.000000 | 0.00% |
| 20-39 | 0 | 0.000000 | 0.00% |
| 40-79 | 446 | 0.001115 | 100.00% |
| 80-159 | 0 | 0.000000 | 0.00% |
| 160-319 | 0 | 0.000000 | 0.00% |
| 320-639 | 0 | 0.000000 | 0.00% |
| 640-1279 | 0 | 0.000000 | 0.00% |
| 1280-2559 | 0 | 0.000000 | 0.00% |
| 2560-5119 | 0 | 0.000000 | 0.00% |
| 5120-4294967295 | 0 | 0.000000 | 0.00% |

**Figure 5.4: Packet lengths measured with Wireshark.**

The figure 5.5 shows the protocol stack. Thus, we can confirm that the data is transmitted correctly using the protocols described in this thesis. We can see from the figure that the data is wrapped in CoAP packets and UDP is used for transmission underneath on top of IPv4.

| Protocol | % Packets | Packets | % Bytes | Bytes | Mbit/s | End Packets | End Bytes | End Mbit/s |
|---|---|---|---|---|---|---|---|---|
| ▼ Frame | 100.00 % | 474 | 100.00 % | 24718 | 0.000 | 0 | 0 | 0.000 |
| ▼ Ethernet | 100.00 % | 474 | 100.00 % | 24718 | 0.000 | 0 | 0 | 0.000 |
| ▼ Internet Protocol Version 4 | 100.00 % | 474 | 100.00 % | 24718 | 0.000 | 0 | 0 | 0.000 |
| ▼ User Datagram Protocol | 100.00 % | 474 | 100.00 % | 24718 | 0.000 | 0 | 0 | 0.000 |
| ▶ Constrained Application Protocol | 100.00 % | 474 | 100.00 % | 24718 | 0.000 | 202 | 9574 | 0.000 |

**Figure 5.5: Protocol hierarchy**

# 6 Conclusions

Before this thesis work, it was recognized at Ericsson that testing and prototyping in the area of Internet of Things could benefit greatly from the testbed, where hundreds or even thousands of virtual sensors and actuators could be run, generate data and communicate with other components on the network. As a solution, this thesis presents an emulator of virtual IoT devices, which can be used as a testbed. It implements a full CoAP-stack with LWM2M management interface on top of it.

In this thesis, we first introduced the IoT concept, which enables smart objects, such as sensors, to identify and control devices in the physical environment over the Internet. As such small devices are typically constrained in terms of CPU, memory and power, lightweight solutions are needed. IoT heavily resides on the fundamentals of web, which is why we presented RESTful architecture and HTTP along with the IoT communication protocols, CoAP and LWM2M, which are designed for constrained environments. CoAP provides minimal overhead and compact message structure, and thus, is preferred over HTTP in IoT. LWM2M provides simple interfaces allowing management and monitoring of IoT devices. IPSO objects also introduced in the work, were used in the iplementation to represent the sensors and actuators. Finally, the virtualization tools, such as Docker were discusssed. Docker, a platform enabling applications to run seamlessly regardless of the environment was used to virtualize the emuator.

Moreover, we have defined a set of design requirements for the emulator. In terms of functionalities, it needs to support basic operations of LWM2M, such as registration, read, write, execute and obseve/notify. In terms of performance, the most significant requirement relates to scalability. The implemeted emulator was based on two use cases: emulation of multiple simple devices and emulation of a complex system. The former one was about representing simple sensors and actuators of devices called weather observer, radiator, presence detector and light controller with IPSO objects, while the latter one modeled a complex fuel injection system in a efficient way by means of composite IPSO object.

The software applications for the virtual devices, that is the LWM2M clients, were implemented in node.js utilizing software libraries, such as coap-node. As the

LWM2M server we used a java based Leshan demo server. Achieving interoperability between the client and server side implementaions as well as adding new features to clients required making adjustments to the existing libraries and introducing new functions. The Leshan server allows monitoring and managing the devices. After the devices have registered to the server, their sensor/actuator resources can be read, wrote, executed and observed. The devices can also communicate directly using CoAP multicast.

In the evaluation part we confirmed that the emulator of virtual devices did support the basic LWM2M operations. Some of the remaining issues were the missing bootrstrap interface, security features and certain data formats, such as binary TLV format that are necessary in real IoT scenarios. The scalability tests revealed the limits of the emulator performnace: the memory usage of the Docker containers was vast, and starting over 600 devices simultaneously couldn't be managed with the exisiting implementation. For that reason, the emulator was optimized by limiting the memory usage of containers to 4 Mbytes and writing a script that allows starting the containers in steps with 100 devices at a time. This improved the performańce by increasing the maximum number of registered devices close to 1000. However, they were managable only with unicast messages. Multicast messages caused the system load to exeed the limit resulting in unstable performance. We also measured bandwidth usage and packet lengths in the network, which met the expectations; traffic generated by the virtual IoT devices was uneven flow of small messages that caused only small peaks in the bandwidth usage at times.

### Future Work

As this thesis provides a simple emulator of virtual IoT devices with basic CoAP and LWM2M features, there are a lot ways to develop it further. Some of the ideas are presented below:

- Security: In real IoT senarios, security modes should be provided. The future work could be implementing the DTLS layer for securing the UDP transmission.
- MQTT: For testing different IoT protocols, MQTT version could be implemented alongside to CoAP. That would enable comparing the protocols in the same scenarios.
- Emulated Environment: Currently the data generated by the virtual devices is just simulated random data. In future, the physical environment, from where

the real sensors collect data, could be emulated. This would require for example creating an environment object that mimics the real environment and from where the virtual sensors read the data.

- Resource Discovery/Lookup Interface: Instead of using multicast for device-to-device communication, it would be more efficient to use resource discovery to find resources in the network or lookup interface that allows querying for certain resources from resource directory.

# References


[1] Höller, J., Tsiatsis, V., Mulligan, C., Avesand, S. et al. (2014) *From Machine-to-Machine to the Internet of Things: Introduction to a New Age of Intelligence.* UK: Elsevier.

[2] Ersue, M. & Keranen, A. (2014) *Terminology for Constrained-Node Networks*, RFC 7228. Available from: https://tools.ietf.org/html/rfc7228. [Accessed on...]

[3] Gubbi, J., Buyya, R., Marusic, S. & Palaniswami, M. (2015) *The Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions*

[4] Chase, J. (2013) *The Evolution of the Internet of Things*, White Paper.

[5] Minerva, R., Biru, A. & Rotondi, D. (2015) *Towards a definition of the Internet of Things (IoT)*, IEEE

[6] International Telecommunication Union (2012) *Series Y: Global Information Infrastructure, InternetProtocol Aspects and Next-generation Networks*, Recommendation ITU-T Y.2060

[7] Jara, A.J., Ladid, L. & Skarmeta, A. () "The Internet of Everything through IPv6: An Analysis of Challenges, Solutions and Opportunities". *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, 4 (3): 97-118.

[8] Khan, R., Khan, U.K., Zaheer, R. & Khan, S. (2012) 'Future Internet: The Internet of Things Architecture,Possible Applications and Key Challenges'. In: *Proceedings of the 10th International Conference on Frontiers of Information Technology.*; Islamabad, Pakistan: December 2012.

[9] Rose, K., Eldridge, S. & Chapin, L./Internet Society (2015) *The Internet of Things: an Overview.*

[10] Kovatsch, M., Lanter, M. & Shelby, Z. (2014) 'Californium: Scalable Cloud Services for the Internet of Things with CoAP'. In: *Proceedings of the 4th International Conference on the Internet of Things*; Cambridge, MA, USA: October 2014.

[11] Kushalnagar, N., Montenegro, G. & Schumacher, C. (2007) *IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals*, RFC 4919. Available from: https://tools.ietf.org/html/rfc4919. [Accessed on...]

[12] Rao, S., Chendanda, D., Deshpande, C. & Lakkundi, V. (2015) 'Implementing LWM2M in Constrained IoT Devices'. In: *Proceedings of Wireless Sensors (ICWiSe) IEEE Conference*; Malaysia: August 2015.

[13] Vermesan, O., Friess, P. & Guillemin, P. et al. *Internet of Things Strategic Research Roadmap.*

[14] Laine, M. () *RESTful Web Services for the Internet of Things.* Available from: http://media.tkk.fi/webservices/personnel/markku_laine/restful_web_services_for_the_internet_of_things.pdf. [Accessed on...]

[15] Jiménez, J. (2015) *Introduction to IPSO Objects* Available from: https://github.com/jaimejim/iot-playground/blob/master/Documentation/IPSO/IPSO_Intro_IOTSHOK.pdf

[16] Fielding, T. (2000) *Architectural Styles and the Design of Network-based Software Architectures.* PhD Dissertation. Irvine: University of California [published].

[17] Keränen, A. & Kovatsch, M. (2015) *RESTful Design for Internet of Things Systems*, draft-keranen-t2trg-rest-iot-00.

[18] Berners-Lee, T., Fielding, R. & Masinter, L. (2005) *Uniform Resource Identifier (URI): Generic Syntax*, RFC 3986. Available from: https://tools.ietf.org/html/rfc3986.

[19] Shelby, Z., Hartke, K. & Bormann, C. (2014) *The Constained Application Protocol (CoAP)*, RFC 7252. Available from:https://tools.ietf.org/html/rfc7252.

[20] Garrett J.J (2005) *Ajax: A New Approach to Web Applications.* Available from: http://adaptivepath.org/ideas/ajax-new-approach-web-applications/. [Accessed on 10.10.2016]

[21] Hartke, K. (2015) *CoRE Application Descriptions*, draft-hartke-core-apps-01. Available from:https://tools.ietf.org/html/draft-hartke-core-apps-01.

[22] Shelby Z. (2012) *Constrained RESTful Environments (CoRE) Link Format*, RFC 6690. Available from: https://tools.ietf.org/html/rfc6690.

[23] Klas, G., Rodermund, F., Shelby, Z., Akhouri, S. & Höller, J. (2014) *"Lightweight M2M": Enabling Device Management and Applications for the Internet of Things*, White Paper.

[24] Fielding, R., J. Gettys, J., Mogul, J. et al. (1999) *Hypertext Transfer Protocol – HTTP/1.1*, RFC 2616 Available from:http://www.rfc-base.org/txt/rfc-2616.txt.

[25] Dierks, T. & Rescorla, E. (2008) *The Transport Layer Security (TLS) Protocol Version 1.2*, RFC 5246. Available from: https://tools.ietf.org/html/rfc5246.

[26] Freier, A., Karlton, P. & Kocher, P. (2011) *The Secure Sockets Layer (SSL) Protocol Version 3.0*, RFC 6101. Available from: https://tools.ietf.org/html/rfc6101.

[27] Bormann, C., Lemay, S., Tschofenig, H. et al. (2016) *CoAP (Constrained Application Protocol) over TCP, TLS, and WebSockets*, draft-ietf-core-coap-tcp-tls-04. Available from: https://tools.ietf.org/html/draft-ietf-core-coap-tcp-tls-04.

[28] Rescorla, E. & Modadugu, N. (2006) *Datagram Transport Layer Security*, RFC 4347. Available from: https://tools.ietf.org/html/rfc4347.

[29] Shelby, Z. (2012) *Constrained RESTful Environments (CoRE) Link Format*, RFC 6690. Available from: https://tools.ietf.org/html/rfc6690.

[30] Hartke, K. (2015) *Observing Resources in the Constrained Application Protocol (CoAP)*, RFC 7641. Available from: https://tools.ietf.org/html/rfc7641.

[31] ETSI (2011) *Electromagnetic compatibilityand Radio spectrum Matters (ERM); System Reference document (SRdoc): Spectrum Requirements for Short Range Device, Metropolitan Mesh Machine Networks (M3N) and Smart Metering (SM) applications*, ETSI TR 103 055 V1.1.1

[32] Open Mobile Alliance (OMA) (2016) *Lightweight Machine to Machine Technical Specification.*

[33] Open Mobile Alliance (n.d.) *OMNA Lightweight M2M (LWM2M) Object & Resource Registry* Available from: http://technical.openmobilealliance.org/Technical/technical-information/omna/lightweight-m2m-lwm2m-object-registry [Accessed on 7.9.2016]

[34] (2016) Jimenez, J., Koster, M. & Tschofenig, H. (2016) 'IPSO Smart Objects'. In: *Position paper for the IOT Semantic Interoperability Workshop*; San Jose, USA: 17-18 March 2016.

[35] Internet Protocol for Smart Objects (IPSO) Alliance. (2014) *IPSO Smart Object Guideline*.

[36] Aazam, M., Hung, P.P. & Huh, E-N. (2014) 'Cloud of Things: Integrating Internet of Things with Cloud Computing and the Issues Involved'. In: *Proceedings of International Bhurban Conference on Applied Sciences & Technology*; Islamabad, Pakistan: January 2014.

[37] Botta, A., De Donato, W., Persico, V. et al. (2016) 'Integration of Cloud computing and Internet of Things: A survey'. *Future Generation Computer Systems* 56: 684-700.

[38] Ericsson (2016) *Ericsson Mobility Report*, June, 2016. Available from: https://www.ericsson.com/res/docs/2016/ericsson-mobility-report-2016.pdf [Accessed on 15.10.2016]

[39] Docker Inc. (2016) *What is Docker?*. Available from: https://www.docker.com/what-docker#/VM. [Accessed on 31.7.2016]

[40] Docker Inc. (2016) *Learn about images & containers*. Available from: https://docs.docker.com/engine/getstarted/step_two/. [Accessed on 31.7.2016]

[41] (2005) Smith, J.E. & Nair, R. (2005) 'The Architecture of Virtual Machines'. *Computer* [Online], 38 (5): 32-38. Available from: https://www.ece.cmu.edu/~ece845/sp11/docs/smith-vm-overview.pdf. [Accessed on 2.8.2016]

[42] Peltz, C. (2003) 'Web Services Orchestration and Coreography'. *Computer* [Online], 36 (10): 46-52. Available from: http://bbs.w3china.org/dragonstar/papers/Peltz_COMP2003Oct.pdf. [Accessed on 2.8.2016]

[43] Ericsson (2016). *Greenhouse – Ericsson Open Wiki*. [unpublic document]

[44] How a Car Works. (2016) *How Fuel injection System Works*. Available from: http://www.howacarworks.com/basics/how-a-fuel-injection-system-works [Accessed on 9.8.2016]

[45] Desarkar, M. S., Sarkar M., Agarwal, P. et al. (n.d.) *Case study of design of an Engine Control Unit*. Available from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.95.4330&rep=rep1&type=pdf [Accessed on 9.8.2016]

[46] AA1Car Auto Diagnosis Repair Help. (n.d.) *Engine Throttle Position Sensor* Available from: http://www.aa1car.com/library/tps_sensors.htm [Accessed on 10.8.2016]

[47] AA1Car Auto Diagnosis Repair Help. (n.d.) *Mass Airflow MAF Sensors* Available from: http://www.aa1car.com/library/maf_sensors.htm [Accessed on 10.8.2016]

[48] AA1Car Auto Diagnosis Repair Help. (n.d.) *Engine Coolant Sensors* Available from: http://www.aa1car.com/library/coolant_sensors.htm [Accessed on 11.8.2016]

[49] AA1Car Auto Diagnosis Repair Help. (n.d.) *Crankshaft & Camshaft Position Sensors* Available from: http://www.aa1car.com/library/crank_sensors.htm [Accessed on 10.8.2016]

[50] Delphi TechSource (2005) **Typical ECM/PCM Inputs**. Available from: http://go.delphi.com/CS/documents/DPSS_Documents/Education/en-us/Education_English_10989.pdf

[51] AA1Car Auto Diagnosis Repair Help. (n.d.) *Diagnose Fuel Pump* Available from: http://www.aa1car.com/library/fuel_pump_diagnose.htm [Accessed on 15.8.2016]

[52] Docker Inc. (2016) *node*, Official repository Available from: https://hub.docker.com/_/node/

[53] PeterEB (2016) *coap-node*. Available from: https://github.com/PeterEB/coap-node. [Accessed on 31.7.2016]

[54] Matteo Collina (2014) *node-coap* Available from: https://github.com/mcollina/node-coap. [Accessed on 31.7.2016]

[55] PeterEB (2016) *node-shepherd* Available from: https://github.com/PeterEB/coap-shepherd. [Accessed on 4.8.2016]

[56] PeterEB (2016) textitsmartobject Available from: https://github.com/PeterEB/smartobject.. [Accessed on 5.8.2016]

[57] Basir (2015) *node-openweathermap*. Available from: https://github.com/baslr/node-openweathermap. [Accessed on 20.7.2016]

[58] Eclipse (2014) *Leshan*. Available from: https://github.com/eclipse/leshan. [Accessed on 6.8.2016]

[59] Eclipse Foundation (2015) textitLeshan. Available from: https://eclipse.org/leshan/. [Accessed on 6.8.2016]

[60] Linux man page. (2010) *iftop - display bandwidth usage on an interface by host* Available from: https://linux.die.net/man/8/iftop [Accessed on 8.10.2016]

[61] Wireshark Foundation (n.d.) *WireShark* Available from: https://www.wireshark.org/ [Accessed on 8.10.2016]